

Лекции по курсу "Сложность комбинаторных алгоритмов"

Николай Николаевич Кузюрин

7 ноября 2003 г.

Глава 1

Элементы теории сложности

При написании этого раздела использовались материалы из [Lov, H.H96].

1.1 Несложно о сложности. Примеры алгоритмов

Неформальное понятие алгоритма и его сложности. Примеры алгоритмических задач различной сложности.

Понятие алгоритма в математике используется давно, но различные его формализации были предложены только в середине 30-х годов нашего столетия, когда и стала складываться теория алгоритмов.

Классическая теория алгоритмов вообще не интересуется сложностными аспектами (временем решения задач на реальных вычислителях). В рамках классической теории алгоритмов, ставятся и решаются задачи о разрешимости различных задач, однако вычислительная сложность полученных решений принципиально не исследуется.

Однако с практической точки зрения, может не быть никакой разницы между неразрешимой задачей и задачей, решаемой за время $\Omega(\exp(n))$, где n — длина входа. Таким образом реализации алгоритмов на реальных вычислительных машинах обязательно требуют анализа сложности их выполнения. Анализом задач с точки зрения вычислительной сложности занимается раздел теории алгоритмов — *теория сложности вычислений*, активно развивающийся с 50х годов — с момента создания вычислительной техники. Теория сложности вычислений занимает промежуточное положение между строгой математикой и реальным программированием. Для математика это в первую очередь математическая теория, строящаяся на основе фундаментальных понятий полиномиальной вычислимости и полиномиальной сводимости. Для программиста-практика — это набор общих методов, парадигм и конструкций, позволяющий в ряде случаев существенно минимизировать прямолинейный перебор вариантов, а в ряде случаев — показать, что эта задача в рассматриваемой постановке скорее всего неразрешима (и, следовательно, следует искать более реалистичные постановки).

Для подавляющего большинства комбинаторных алгоритмических задач существует простой прямолинейный алгоритм, основанный на *переборе* всех или почти всех вариантов. В ряде случаев (скажем, при небольших размерах входных данных) такой подход вполне удовлетворителен.

Теория сложности вычислений начинается в тот момент, когда прямолинейные (в нашем случае — переборные) алгоритмы становятся неприемлемыми или, по меньшей мере, экономически невыгодными, ввиду роста сложности решаемых оптимизационных задач.

Заметим, что выбор конкретной программной реализации выходит за рамки теории сложности. Поэтому алгоритмы обычно описываются на наиболее примитивном, алголоподобном языке.

1.1.1 Примеры задач на натуральных числах.

Мы начнем рассмотрение парадигмы сложности с простой задачи о натуральных числах.

Итак, даны три натуральных числа x , n , и m . Требуется вычислить $y = x^n \bmod m$. (Деление по модулю здесь добавлено, чтобы не касаться проблем связанных с длиной операндов).

Легко придумать следующий прямолинейный алгоритм (См. алгоритм 1), который в цикле от 1 до n выполняет следующее действие: $y \leftarrow y \cdot x \bmod m$.

Алгоритм 1 Тривиальное вычисление $y = x^n \bmod m$

Вход: Натуральные $n, x, m \geq 1$

Выход: $y = x^n \bmod m$

$y \leftarrow 1$

for all $i \in 1..n$ **do**

$y \leftarrow y \times x \bmod m$ {Напомним, что $x^n \bmod m \equiv (x \bmod m)^n \bmod m$ }

end for

return y

Очевидный анализ показывает сложность этого алгоритма — $O(n)$. Пока, на неформальном уровне, под сложностью мы будем понимать число элементарных операций.

Можно существенно улучшить алгоритм 1, используя соображения, основанные на двоичном разложении показателя степени (См. алгоритм 2).

Действительно, рассмотрев двоичное разложение $n = \sum_{i=1}^k a_i 2^i$, где $a_i \in \{0, 1\}$, заметим, что

$$x^n = x^{\sum_{i=1}^k a_i 2^i} = \prod_{i=1}^k x^{a_i 2^i} = \prod_{\{i: a_i > 0\}} x^{2^i}$$

Т.е. достаточно провести $k \leq (\log_2 n + 1)$ итераций по двоичному разложению n ,

чтобы на каждой i -й итерации, в зависимости от i -го бита в разложении, проводить домножение результата на сомножитель x^{2^i} , который легко получить из сомножителя предыдущей итерации (ключевое соотношение):

$$x^{2^i} \leftarrow (x^{2^{i-1}})^2.$$

Анализ же этого алгоритма (алгоритм 2) показывает что его сложность — $O(\log n)$. Как мы видим разница в сложности двух алгоритмов экспоненциальна!

Заметим, что процедура расчета модульной экспоненты весьма распространена в различных сетевых протоколах, например при установке защищенного интернет-соединения (всякий раз, когда в браузере адрес начинается с **https://**, при установке соединения работают алгоритмы криптографии и вычисляется модульная экспонента). Если взять длину ключа 128-бит (длина показателя экспоненты близка к длине ключа), то получается простое сравнение, количества умножений в обоих алгоритмах:

Длина ключа (бит)	Умножений в алгоритме 1	Умножений в алгоритме 2
56	$\approx 7.2 \cdot 10^{16}$	≈ 56
128	$\approx 3.4 \cdot 10^{38}$	≈ 128

Комментарии излишни.

Алгоритм 2 Разумное вычисление $y = x^n$

Вход: Натуральные $n, x, m \geq 1$

Выход: $y = x^n \bmod m$

```

 $y \leftarrow 1$ 
 $X \leftarrow x$ 
 $N \leftarrow n$ 
while  $N \neq 0$  do
  if  $N \bmod 2 = 0$  then
     $X \leftarrow X \times X \bmod m$ 
     $N \leftarrow N/2$ 
  else
     $y \leftarrow y \times X \bmod m$ 
     $N \leftarrow N - 1$ 
  end if
end while
```

Теперь рассмотрим примитивный алгоритм точного вычисления факториала (тоже по модулю m) (см. алгоритм 3). Нетрудно видеть, что его сложность — $O(n)$. Можно ли ее существенно улучшить, как в случае с возведением в степень? Оказывается, на сегодняшний день это является известной открытой проблемой (См. следствия из **Problem 4** в [Sma00]).

Рассмотрим еще одну очень важную задачу, связанную с возведением в степень по модулю.

Задача 1 Дискретный логарифм.

Пусть p – нечетное простое число. Известны натуральные a и b и выполняется соотношение

$$a^x \equiv b \pmod{p}$$

Найти x .

По сути эта обратная задача к задаче возведения в степень по модулю. Можем ли мы как и в предыдущем примере ожидать легкого решения? Оказывается, нет, вычисление дискретного логарифма является очень сложной задачей и самые быстрые из известных алгоритмов требуют экспоненциального (по длине двоичной записи p) времени. На этом важном свойстве (свойстве *односторонней* вычислимости модульной экспоненты) основано множество алгоритмов современной криптографии.

Таким образом, мы увидели, что несмотря на внешнюю похожесть, сложность похожих задач может существенно отличаться. Для обнаружения и понимания такой разницы и нужен анализ сложности, являющийся предметом нашего курса.

Алгоритм 3 Вычисление факториала $y = n! \pmod{m}$

Вход: Натуральные $n, m \geq 1$

Выход: $y = n! \pmod{m}$

```

 $y \leftarrow 1$ 
for all  $i \in 1..n$  do
   $y \leftarrow y \times i \pmod{m}$ 
end for
return  $y$ 

```

Алгоритм Евклида. Рассмотрим одну из классических задач — нахождение наибольшего общего делителя двух целых чисел.

Вход: Неотрицательные целые числа a и b .

Выход: $x = \text{НОД}(a, b)$.

Мы рассмотрим сейчас алгоритм Евклида для решения этой задачи и проанализируем его. Без ограничения общности будем полагать, что $a \leq b$.

Евклид: 1) $a = 0$. Тогда $\text{НОД}(a, b) = b$.

2) $a \neq 0$. Тогда разделим a на b с остатком, т.е. найдем l и r такие, что $b = la + r$, $0 \leq r < a$.

Утверждение: $\text{НОД}(a, b) = \text{НОД}(a, r)$.

Доказательство. Очевидно.

Утверждение: Время работы алгоритма Евклида $O(\log a + \log b)$ арифметических операций над натуральными числами.

Доказательство. Имеем: $b > a + r > 2r$. Отсюда получаем, что $r < b/2$. Таким

образом, $ar < ab/2$ и после $\lceil \log(ab) \rceil$ итераций произведение натуральных чисел станет меньше 1. А это означает, что одно из них равно нулю (т.е. НОД уже найден).

Отметим, что длина записи исходных данных в этой задаче есть $O(\log a + \log b)$ и по порядку совпадает с числом операций алгоритма Евклида (алгоритм является эффективным).

Отметим, что модификация алгоритма, где деление с остатком заменяется на вычитание меньшего числа из большего, не является столь эффективным. Более точно, после одной итерации мы заменяем поиск НОД(a, b) на поиск НОД($a, b - a$). Если b много больше a , число операций вычитания пропорционально b/a (а не сумме их логарифмов как в первоначальном алгоритме).

1.1.2 Приближенные алгоритмы. Многопроцессорные расписания

Не все алгоритмы находят точное решение. Некоторые довольствуются нахождением приближенного (в некотором смысле) решения. Такие алгоритмы называются приближенными. Для них актуальной задачей является анализ точности получаемого решения.

Не давая пока формальных определений рассмотрим для иллюстрации одну из простейших задач составления расписаний.

Имеется m одинаковых машин и n независимых работ с заданными длительностями исполнения t_1, \dots, t_n . Требуется распределить эти работы по машинам так, чтобы минимизировать максимальную загрузку (загрузка машины равна сумме длительностей работ, приписанных данной машине).

Несмотря на простоту постановки эта задача трудна с вычислительной точки зрения (NP-полна). Традиционный подход к таким задачам состоит в использовании простых эвристик.

Мы рассмотрим сейчас одну из таких эвристик и проанализируем ее.

Эвристика: Берется произвольная работа и помещается на машину, имеющую наименьшую загрузку.

Утверждение: Построенное расписание отличается от оптимального (по критерию минимизации максимальной загрузки) не более, чем в два раза.

Доказательство. Пусть $t_{max} = \max_i t_i$, T_0 – длина оптимального расписания (сумма длин работ на наиболее загруженной машине), T_A – длина расписания, которое построено нашей эвристикой. Нам потребуется одно важное свойство расписаний, которые строятся нашей эвристикой:

Свойство: после каждого шага разница в загрузке между наиболее и наименее загруженными машинами не превосходит t_{max} .

Обозначим через T_{min} – сумму длин работ на наименее загруженной машине

(в расписании, построенном нашей эвристикой). Имеют место простые неравенства: $T_0 \geq T_{min}$, $T_0 \geq t_{max}$.

А теперь мы легко можем оценить качество получаемого расписания:

$$\frac{T_A}{T_0} \leq \frac{T_{min} + t_{max}}{T_0} \leq \frac{T_0 + t_{max}}{T_0} \leq 1 + \frac{t_{max}}{T_0} \leq 1 + 1 = 2.$$

Следует подчеркнуть, что эвристики очень часто используются для решения задач на практике. В рассмотренном примере алгоритм (эвристика) обладал одним дополнительным свойством: он применим и в случае когда работы поступают одна за другой, поскольку решение о назначении машины для данной работы принимается только на основании информации о состоянии машин (это так называемые on-line алгоритмы).

1.1.3 Примеры задач на графах. Кратчайшие пути и задача коммивояжера.

Теперь рассмотрим две классические задачи на графах.

Задача 2 КОММИВОЯЖЕР¹

Заданы n городов v_1, v_2, \dots, v_n и попарные расстояния $d_{ij} \equiv d(v_i, v_j)$ между ними, являющиеся положительными целыми числами.

Чему равна наименьшая возможная длина кольцевого маршрута, проходящего по одному разу через все города (см. Рис. 1.1)? Иными словами, требуется найти минимально возможное значение суммы

$$\sum_{i=1}^{n-1} d_{\pi(i), \pi(i+1)} + d_{\pi(n), \pi(1)}, \quad (1.1)$$

где минимум берется по всем перестановкам $\pi(1), \dots, \pi(n)$ чисел $1, \dots, n$.

Задача 3 КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ²

Заданы n вершин графа (узлов сети) v_1, v_2, \dots, v_n и положительные целые длины дуг $d_{ij} \equiv d(v_i, v_j)$ между ними.

Чему равна наименьшая возможная длина пути, ведущего из v_1 в v_n ? Иными словами, чему равно минимально возможное значение суммы

$$\sum_{i=1}^{\ell-1} d(v_{\sigma(i)}, v_{\sigma(i+1)}), \quad (1.2)$$

¹В англоязычной литературе — Traveling Salesman Problem, сокращенно TSP.

²В англоязычной литературе — Shortest Path Problem.

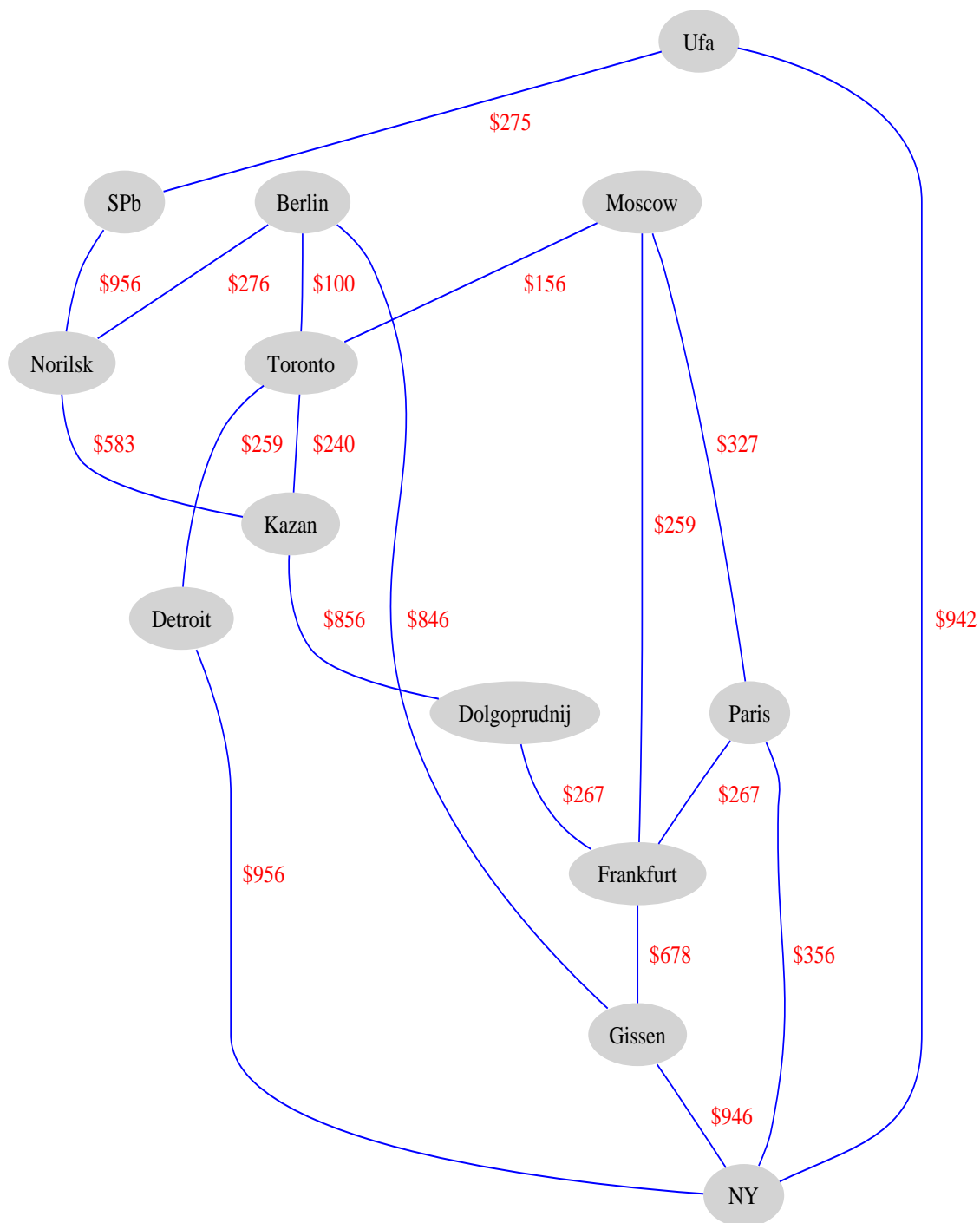


Рис. 1.1: Иллюстрация типичных исходных данных для задачи КОММИВОЯЖЕР

Число городов, n	Число циклов в переборном алгоритме, $n!$
5	120
8	40320
10	$3,6 \cdot 10^6$
13	$6,2 \cdot 10^9$
15	$1,3 \cdot 10^{12}$
30	$2,7 \cdot 10^{32}$

Таблица 1.1: Значения функции $n!$

где минимум берется по всем числовым последовательностям

$$\sigma(1), \dots, \sigma(\ell)$$

(на этот раз не обязательно длины n), в которых $\sigma(1) = 1$ и $\sigma(\ell) = n$.

Переборный алгоритм для задачи о коммивояжере просто перебирает все возможные перестановки π (см. алгоритм 4).

Алгоритм 4 Переборный алгоритм для КОММИВОЯЖЕР

Вход: $m > 0$, $d_{ij} > 0$, $\forall i, j \in (1, \dots, n)$

Выход: Перестановка π_{opt} , такая, что (1.1) минимальна.

```

 $Sum_{best} \leftarrow +\infty$ 
for all  $\pi \in \pi(1, \dots, n)$  do
     $Sum \leftarrow (1.1)$ 
    if  $Sum_{best} > Sum$  then
         $Sum_{best} \leftarrow Sum$ 
         $\pi_{opt} \leftarrow \pi$ 
    end if
end for
return  $\pi_{opt}$ 

```

Переборный алгоритм для задачи о кратчайшем пути устроен совершенно аналогичным образом.

При анализе сложности алгоритма 4 видно, что вычисление индивидуальной суммы (1.1) не представляет особых трудностей; проблема состоит в том, что этот процесс придется повторить слишком много, $n!$ раз. Некоторые значения факториала приведены в таблице 1.1.

Видно, что при $n = 5$ расчет всех вариантов согласно переборному алгоритму может быть произведен вручную. При $n = 8$ для его проведения в разумный отрезок времени нужно привлечь программируемый калькулятор, а при $n = 10$ — уже более быстродействующую вычислительную технику. Когда число городов доходит до 13, потребуется суперкомпьютер, а случай $n = 15$ выходит за пределы возможностей

любой современной вычислительной техники. Число возможных вариантов при $n = 30$ превышает количество атомов на Земле.

Все это в равной мере приложимо к переборному алгоритму для решения задачи о кратчайшем пути и вообще практически любой задачи дискретной оптимизации.

Разумеется, наиболее радикальное и предпочтительное решение состоит в том, чтобы *разработать точный эффективный алгоритм для решения нашей задачи*, исключающий (или, по меньшей мере, минимизирующий) непосредственный перебор вариантов.

Рассмотрим в качестве типичного примера задачу о кратчайшем пути в графе и построим для нее алгоритм динамического программирования. Заметим, что различные варианты разработанного алгоритма реально используется при маршрутизации компьютерных сетей³.

Предположим, что в каждом узле v_i находится пара $P_i \equiv (j, D_i)$, служащая направлением и оценкой стоимости наиболее дешевого пути из v_i до v_n . Изначально, во всех узлах кроме узлов-соседей v_n все эти пары установлены в $(Null, +\infty)$ (на самом деле $+\infty$ — выбранное положительное целое число, по порядку большее фигурирующих стоимостей ребер).

А в каждом v_j — узле-соседе v_n , эта пара определена в (n, d_{jn}) — прямой переход в узел v_n .

Затем мы пытаемся улучшить эти оценки. На каждой итерации, каждый узел v_i обращается к каждому соседнему узлу v_j , за информацией о самом дешевом пути до v_n , который тот ему может предоставить на данный момент, и для своей пары выбирает того соседа v_j , который обеспечивает $\min_j d_{ij} + D_j$ и устанавливает свою i -ю пару в (j, D_i) , где $D_i = d_{ij} + D_j$.

Оказывается, что если мы повторим эти итерации n раз подряд, то полученные в итоге оценки D_i совпадут с кратчайшими расстояниями из v_i в v_n (См. алгоритм 5).

Упражнение 1 Докажите математическую корректность алгоритма 5.

Очевидно, что вычислительная сложность алгоритма 5 — $O(n^3)$. Таким образом, методы динамического программирования (о них в следующих лекциях), позволили построить *эффективный* (более детально о понятии эффективности тоже потом) алгоритм, для задачи, очевидным решением которой был полный перебор, требующий экспоненциального времени.

Задачи КОММИВОЯЖЕР и КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ чрезвычайно похожи по своей структуре, и мы постарались подчеркнуть это сходство в их формулировках.

Таким образом, сравнительно успешно устранив перебор для задачи КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ с помощью динамического программирования, естественно

³См. например стандарты OSPF, Open Shortest Path First Routing Protocol, RFC 2740

Алгоритм 5 Алгоритм КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ в одну вершину

Вход: $n > 0$, $d_{ij} > 0$, $\forall i, j \in (1, \dots, n)$ **Выход:** Кратчайший путь из любой вершины v_i в v_n .

```

for all  $i \in 1..n$  do
   $P_i \equiv (Null, +\infty)$  {сбрасываем начальные оценки}
end for
for all  $j \in \{j : d_{jn} < +\infty\}$  do
   $P_j \equiv (n, d_{jn})$  {устанавливаем оценки для соседей}
end for
for all  $iteration \in 1..n$  do {достаточно, чтобы процесс сошелся}
  for all  $i \in 1..n$  do {по всем узлам}
    for all  $j \in \{j : d_{ij} < +\infty\}$  do {по всем соседям  $v_i$ }
      if  $D_i > D_j + d_{ij}$  then { $j$ -й сосед предлагает нам лучший путь}
         $D_i \leftarrow D_j + d_{ij}$  {улучшаем оценку}
         $P_i = (j, D_i)$  {направляем путь к  $j$ -му соседу}
      end if
    end for
  end for
end for
return  $\forall i \ (P_i(1), P_{P_i(1)}(1), P_{P_i(P_i(1))}(1), \dots, n)$  {От всех узлов прокладываем путь к  $v_n$ }

```

задаться аналогичным вопросом для КОММИВОЯЖЕРА. Довольно быстро оказывается, впрочем, что ни один из "естественных" методов сокращения перебора к последней задаче неприменим. Таким образом, встает законный вопрос, а можно ли вообще решить задачу КОММИВОЯЖЕР с помощью точного алгоритма, существенно более эффективного, нежели переборный?

Одним из главных достижений теории сложности вычислений является стройная и элегантная теория **NP-полноты**, позволяющая в 99% случаев дать вполне удовлетворительный ответ на этот вопрос.

Эта теория будет рассмотрена далее, пока же термин **NP-полная задача** можно понимать неформально в смысле *труднорешаемая переборная задача, для которой существование алгоритма намного более эффективного нежели простой перебор вариантов, крайне маловероятно*.

В частности, в одном из первых исследований было показано, что задача КОММИВОЯЖЕР **NP-полна**, и тем самым на возможность построения для нее точного эффективного алгоритма рассчитывать не приходится.

Соответственно этому, следующий вопрос, на который пытается ответить теория сложности вычислений — это какие рекомендации можно дать практическому разработчику алгоритмов в такой ситуации, т.е. в тех случаях, когда результаты диагностики интересующей его задачи на существование для нее точных эффективных алгоритмов столь же неутешительны, как в случае задачи КОММИВОЯЖЕР. Одна из таких рекомендаций состоит в следующем: попытаться *проанализировать*

постановку задачи и понять, нельзя ли *видоизменить ее формулировку* так, чтобы с одной стороны новая формулировка все еще была бы приемлема с точки зрения практических приложений, а с другой стороны — чтобы в этой формулировке задача уже допускала эффективный алгоритм. Кстати, в качестве побочного продукта в ряде случаев такой сложностной анализ позволяет лучше понять природу задачи уже безотносительно к ее вычислительной сложности.

Например, пусть некий начинающий проектировщик сетей задумал спроектировать оптимальную компьютерную сеть, соединяющую n корпусов общежитий. Он только что изучил кольцевые топологии сети и вознамерился проложить кольцевой сетевой маршрут через все корпуса общежитий. Стоимость прокладки кабеля между любыми двумя корпусами известна (Если между какими-то корпусами кабель проложить нельзя, например из-за постоянных работ по ремонту теплотрасс, то стоимость полагается равной $+\infty$). Формулировка этой задачи в чистом виде совпадает с задачей КОММИВОЯЖЕР. Как мы уже видели раньше, если число корпусов больше 10, то проектировщику потребуется доступ к дорогой вычислительной технике, а если больше 15, то гарантированное получение решения за время жизни проектировщика пренебрежимо маловероятно.

Что же делать незадачливому проектировщику сети? Почитав дальше книгу по проектированию сетей, он решает построить *минимальную связную сеть*, используя минимальные связные (из $n - 1$ дуги) подграфы исходного потенциального графа связности общежитий, так называемые *остовные деревья*⁴.

Задача 4 МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО ⁵

Задан связный неориентированный граф $G = (V, E)$, где V — множество вершин, $|V| = n$, E — множество ребер между ними, и весовая функция $w : E \rightarrow \mathbb{Z}^+$. Иными словами, есть n вершин v_1, \dots, v_n и положительные целые веса дуг $w_{ij} \equiv w(v_i, v_j)$ ⁶ между ними.

Чему равен наименьший возможный вес остовного дерева? Т.е., требуется найти минимально возможное значение суммы

$$\sum_{(i,j) \in T} w(v_i, v_j), \quad (1.3)$$

где минимум берется по всем остовным деревьям на n вершинах, т.е. по всем множествам T из $(n - 1)$ дуг, связывающим все n вершин в единую сеть.

На первый взгляд, переход от задачи КОММИВОЯЖЕР к задаче МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО только увеличивает трудности, стоящие перед нашим проектировщиком: перебор по множеству всех замкнутых путей заменяется перебором по еще более необозримому множеству произвольных остовных деревьев.

⁴В англоязычной литературе — spanning trees

⁵В англоязычной литературе — Minimum Spanning Tree

⁶Можно вводить веса на ребрах, как w_e , $e \in E$

Тем не менее, эта интуиция в данном случае в корне ошибочна: эффективные алгоритмы для задачи МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО существуют.

Опишем один из них, так называемый *алгоритм Прима* (Prim)⁷ [Том99]. В этом алгоритме минимальный остов строится постепенно: сначала выбирается произвольная вершина, которая включается в остов, затем, на каждой итерации, к текущему остову добавляется наиболее дешевое ребро (u, v) , соединяющее какую-либо вершину из остова u , с какой-либо вершиной v не из остова.

Ниже (См. Рис. 1.2) показана эволюция этого алгоритма — стартовое состояние, после 5 шага, после 8, и конечное остовное дерево.

Алгоритм 6 Упрощенный алгоритм Прима

Вход: $m > 0$, $w_{ij} > 0$, $\forall i, j \in (1, \dots, n)$ {Список ребер задается матрицей связности $\{w_{ij}\}$ — не оптимальный вариант}

Выход: Остов A минимального веса.

$A \leftarrow \{1\}$ {можно выбрать произвольную вершину}

for all $iteration \in 1..n$ **do** {достаточно, чтобы процесс сошелся}

$W_{best} \leftarrow +\infty$

$V_{best} \leftarrow null$

for all $i \in \{1..n : i \notin A\}$ **do** {по всем не включенным в A узлам}

for all $j \in A$ **do** {по всем включенным в A узлам}

if $w_{ij} < W_{best}$ **then** { w_{ij} более дешевая дорога из остова}

$W_{best} \leftarrow w_{ij}$ {улучшаем оценку}

$V_{best} = i$ {выбираем v_i для добавления}

end if

end for

end for

$A \leftarrow A \cup V_{best}$ {добавляем вершину с самым дешевым путем до остова}

end for

return A {Индексы вершин минимального остова G }

Упражнение 2 Докажите корректность алгоритма 6.

Сложность этого примитивного варианта алгоритма Прима (алгоритм 6) очевидно равна $O(n^3)$, а не экспоненциальна, как в алгоритме для задачи КОММИ-ВОЯЖЕР. На самом деле, если использовать представление графа в виде списка вершин и ребер, и специальные структуры данных, то можно уменьшить сложность этого алгоритма до $O(|E| + |V| \log |V|)$.

Упражнение 3 Придумайте такой алгоритм.

Этот крайне поучительный пример наглядно демонстрирует, что при устранении перебора внутренняя структура задачи имеет гораздо большее значение, нежели размер последнего и что внешность задачи во многих случаях бывает крайне

⁷Существует еще алгоритм Краскала (Kruskal) [Том99].

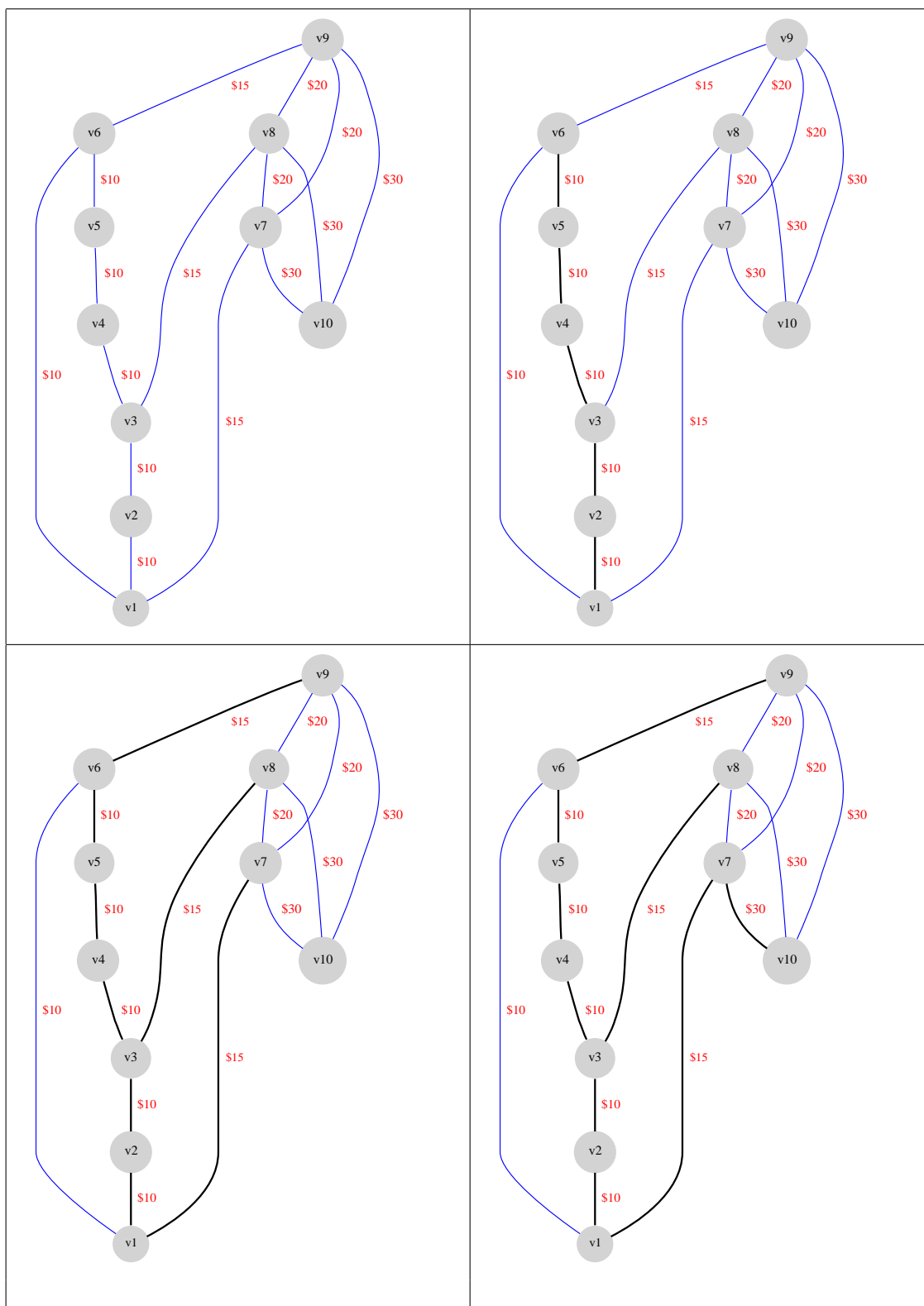


Рис. 1.2: Иллюстрация работы алгоритма 6

обманчивой, и никакие правдоподобные рассуждения "по аналогии" не могут заменить настоящего сложностного анализа. Последний в ряде случаев приводит к довольно неожиданным выводам, зачастую противоречащими обыкновенной интуиции.

Перечислим основные выводы из этой вводной лекции:

- Для подавляющего большинства задач дискретной оптимизации существует тривиальный алгоритм, основанный на переборе всех вариантов. Переборные алгоритмы становятся нереализуемыми с практической точки зрения уже при сравнительно небольшом размере входных данных (см. таблицу 1.1).
- Теория сложности вычислений занимается построением и анализом *эффективных* алгоритмов — в данном случае алгоритмов, в которых перебор вариантов устранен или по крайней мере сокращен до приемлемого уровня.
- Основное внимание концентрируется на сравнительно небольшом числе "модельных", классических алгоритмических задач, каждая из которых описывает огромное число самых разнообразных приложений.
- Наиболее предпочтительным решением является построение точного эффективного алгоритма для рассматриваемой задачи.
- Имеется обширный класс **NP**-полных задач, для которых существование точного эффективного алгоритма представляется крайне маловероятным. Классификация переборных задач на **NP**-полные и тех, которые поддаются решению с помощью точного эффективного алгоритма, оказывается весьма успешной — под нее подпадает подавляющее большинство переборных задач.
- В тех случаях, когда интересующая разработчика практических алгоритмов задача оказывается **NP**-полной, имеет смысл попробовать построить эффективный алгоритм для какой-либо ее модификации, приемлемой с практической точки зрения.
- В тех случаях, когда такую модификацию найти не удастся, имеет смысл попробовать построить для решения задачи *приближенный* эффективный алгоритм, который гарантирует нахождение решения, приближающегося к оптимальному с некоторой наперед заданной точностью.
- Рекомендации, даваемые теорией сложности вычислений в чисто прагматических целях построения эффективных алгоритмов, зачастую позволяют лучше понять внутреннюю природу самой задачи.

1.1.4 Задачи сортировки и поиска

Задача сортировки заключается в следующем.

Имеется произвольный массив $A : a_1, \dots, a_n$. Требуется путем сравнений отсортировать этот массив таким образом, чтобы элементы расположились в порядке возрастания (или убывания), то есть

$$a_{i1} \leq a_{i2} \leq \dots \leq a_{in}.$$

Медиана массива

Имеется неупорядоченный массив длины n .

Если n нечетно и $m = \lceil n/2 \rceil$, то m -й в порядке убывания элемент называется медианой.

Сортировка слиянием

Слияние – это объединение двух или более упорядоченных массивов в один упорядоченный.

Пусть требуется упорядочить массив по убыванию. Для этого сравниваются два наименьших элемента обоих массивов и наименьший из них выводится как наименьший элемент суммарного массива, затем процедура повторяется.

Нетрудно видеть, что слияние двух упорядоченных последовательностей длин m и n происходит за $O(n + m)$ сравнений.

Алгоритм сортировки слиянием

Исходный массив длины n делится пополам. Затем производится рекурсивная сортировка каждой половинки и их слияние. Пусть $T(n)$ – число сравнений, достаточное для сортировки слиянием.

Можем записать рекуррентное соотношение:

$$T(n) \leq 2T(n/2) + O(n).$$

Чтобы решить это неравенство применим метод подстановки. Проверим, что $T(n) = cn \log n$ является решением при подходящей константе c . Имеем:

$$cn \log n \leq cn \log(n/2) + O(n) = cn \log n - cn + O(n),$$

и при достаточно большом значении c неравенство выполнено. Таким образом, для сортировки массива из n элементов достаточно $cn \log n$ сравнений.

Рекурсивный алгоритм нахождения медианы за линейное время

Имеется неупорядоченный массив длины n . Пусть $n = 10t$. Обозначим через $T(n)$ время необходимое для нахождения медианы в массиве из n элементов.

1) Разобьем массив на группы по пять элементов. Получим $2t$ групп. Отсортируем каждую группу за константное время и возьмем в каждой группе медиану a_i . Это потребует не более cn сравнений, где c – некоторая константа.

2) Рекурсивно найдем медиану массива $\{a_i\}$, состоящего из $2t$ медиан, обозначим ее x . Это потребует не более $T(n/5)$ сравнений.

3) $n - 1$ элементов, отличных от x , разбиваются на два множества:

$$I = \{i : a_i \leq x\}$$

$$J = \{j : a_j > x\}$$

Факт. Мощность каждого из множеств не больше $\frac{7}{10}n$ элементов.

Это вытекает из следующих наблюдений:

Из пяти элементов каждой группы по крайней мере 3 элемента меньше или равны a_i . Всего групп $2t$.

По крайней мере t элементов из множества медиан a_i меньше или равны x .

Таким образом $3t$ элементов меньше или равны x . $3t = \frac{3}{10}n$.

4) Мы нашли r элементов, больших x , и $n - 1 - r$ элементов, меньших x . Если $m = r + 1$, то x – искомая медиана.

Если $m < r + 1$, то рекурсивно ищем m -й элемент в порядке убывания из r больших элементов.

Если $m > r + 1$, то рекурсивно ищем $(m - 1 - r)$ -й элемент в порядке убывания из $n - r - 1$ меньших элементов.

И n , и $n - r - 1$ оба меньше или равны $\frac{7}{10}n$, поэтому этот шаг потребует не более $\frac{7}{10}cn$ сравнений. $c = const$.

Оценим время работы алгоритма. Имеет место следующее неравенства:

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$$

$c = const$, $T(n)$ – время поиска медианы в массиве из n элементов. Неравенство выполняется поскольку $9/10 < 1$. Действительно, подставляя $T(n) = c_1n$ получим:

$$c_1n \leq cn + c_1\frac{n}{5} + c_1\frac{7}{10}n = cn + c_1(9/10)n.$$

Очевидно это неравенство выполнено при достаточно большом значении c_1 . Следовательно, описанный алгоритм работает линейное время.

Вероятностные алгоритмы нахождения медианы и сортировки

Быстрая сортировка.

В алгоритме быстрой сортировки выбирается случайный элемент массива и с ним сравниваются все остальные элементы. Получаются два новых массива (больших и меньших случайного элемента), к которым алгоритм применяется рекурсивно. Обозначим через $T(n)$ – среднее время быстрой сортировки (т.е. число сравнений).

Имеем рекуррентное соотношение следующего типа:

$$T(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + O(n).$$

Гипотеза: $T(n) \leq an \log n + b$, $a > 0$, $b > 0$ – константы.

Проверим:

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=1}^{n-1} (ak \log k + b) + O(n) = \\ &= \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + \frac{2b}{n} (n-1) + O(n) \leq \\ &\leq \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + 2b + O(n). \end{aligned}$$

Имеем:

$$\begin{aligned} \sum_{k=1}^{n-1} k \log k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \leq \\ &(\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \\ \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k &\leq \frac{1}{2} n(n-1) \log n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \leq \\ &\frac{1}{2} n^2 \log n - \frac{1}{8} n^2. \end{aligned}$$

Подставляя в рекуррентное соотношение, получаем:

$$\begin{aligned}
T(n) &\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + 2b + O(n) \leq \\
&\leq an \log n - \frac{a}{4} n + 2b + O(n) \leq \\
&\leq an \log n + b
\end{aligned}$$

при достаточно больших n , $a \left(\frac{a}{4} n > b + O(n) \right)$

Вероятностный алгоритм нахождения медианы

Вероятностный алгоритм нахождения медианы очень похож на алгоритм быстрой сортировки. В нем также выбирается случайный элемент массива и с ним сравниваются все остальные элементы. Получаются два новых массива (больших и меньших случайного элемента). Однако далее, алгоритм применяется рекурсивно лишь к одному из них.

Упражнение. Почему достаточно просмотреть лишь один из массивов? Опишите детали алгоритма.

Обозначим через $T(n)$ – среднее время работы описанного алгоритма (т.е. число сравнений).

Имеем следующее рекуррентное соотношение :

$$\begin{aligned}
T(n) &\leq \frac{1}{n} \sum_{k=1}^{n-1} T(\max\{k, n-k\}) + O(n) \leq \\
&\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n).
\end{aligned}$$

Гипотеза: $T(n) \leq cn$, $c > 0$ – константа.

Проверим:

$$\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) \leq \\
&\frac{2c}{n} \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + O(n) = \\
&\frac{2c}{n} \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + O(n) =
\end{aligned}$$

$$\begin{aligned}
&= \frac{2c}{n} \left(\frac{1}{2}n(n-1) - \frac{1}{2}(\lceil n/2 \rceil - 1)\lceil n/2 \rceil \right) + O(n) \leq \\
&\leq c(n-1) - \frac{c}{2}(n/2 - 1) + O(n) = \\
&= c\frac{3n}{4} - c + O(n) \leq cn
\end{aligned}$$

при $\frac{c}{4}n - c > O(n)$.

1.1.5 Еще о поиске кратчайших путей

Рассмотрим снова задачу о кратчайшем пути в графе и опишем эффективные алгоритмы ее решения.

Кратчайшие пути между всеми парами вершин в ориентированном графе.

Пусть ребрам графа приписаны веса (возможно и отрицательные), причем в графе нет ориентированных циклов отрицательной длины.

Пусть d_{ij}^k обозначает кратчайшее расстояние между вершинами i и j по всем путям, чьи промежуточные вершины принадлежат множеству $\{1, 2, \dots, k\}$. Пусть P – кратчайший среди таких путей.

Случай 1. k не является промежуточной вершиной пути P . Тогда $d_{ij}^k = d_{ij}^{k-1}$

Случай 2. k является промежуточной вершиной пути P . Тогда $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$.

Утверждение: d_{ij}^n – кратчайшее расстояние от вершины i до вершины j .

Доказательство. Очевидно.

Алгоритм Флойда-Уоршелла.

Цикл по k от 1 до n

 Цикл по i от 1 до n

 Цикл по j от 1 до n выполнить

$$d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}).$$

Очевидно, что вычислительная сложность алгоритма — $O(n^3)$.

Кратчайшие пути от одной вершины в ориентированном графе. Алгоритм Дейкстры

Рассматриваются ориентированные графы $G = (V, E)$ с неотрицательными весами на ребрах, т.е. каждому ребру $(u, v) \in E$ приписан вес $w(u, v) \geq 0$.

Имеется выделенная вершина (источник) и мы хотим найти кратчайшие расстояния от нее до каждой из остальных вершин графа. Обозначим кратчайшее расстояние от s до v через $\delta(s, v)$.

Алгоритм Дейкстры

- 1) Полагаем $S = \emptyset$, $d[s] = 0$, $d[v] = +\infty$, для всех $v \neq s$.
- 2) Если $S \neq V$, добавляем к S вершину v , с минимальным значением $d[v]$, иначе STOP.
- 3) Пересчитываем значения $d[u]$ для всех вершин u смежных с v по формуле: $d[u] = \min(d[u], d[v] + w(u, v))$ и возвращаемся на шаг 2).

Оказывается, что после завершения работы алгоритма, полученные в итоге оценки $d[v]$ совпадут с кратчайшими расстояниями $\delta[s, v]$.

Доказательство корректности. От противного. Пусть u – первая вершина, для которой $d[u] \neq \delta(s, u)$ в процессе расширения S . Рассмотрим кратчайший путь P из s в u . В этом пути найдем первую вершину (обозначим ее y), не принадлежащую множеству S . Пусть x непосредственно предшествует вершине y в P . В силу неотрицательности весов имеем: $\delta(s, y) \leq \delta(s, u)$. Имеем также: $d[x] = \delta(s, x)$ (это выполнено по предположению для всех элементов S). Тогда по правилу пересчета $d[y] = \delta(s, y)$, т.к. $\delta(s, y) = \delta(s, x) + w(x, y)$. Таким образом,

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u],$$

но по правилу выбора вершины мы имели $d[u] \leq d[y]$. Значит, $d[u] = d[y]$, что дает $d[u] = \delta(s, u)$ и противоречит выбору u .

Очевидно, что простая верхняя оценка сложности алгоритма Дейкстры есть $O(n^2)$, однако она может быть доведена до $O(m \log m)$ с помощью специальных структур данных (здесь $|V| = n$, $|E| = m$).

1.2 Формально об алгоритмах

Формальное определение алгоритма. Класс вычислимых функций. Существование алгоритмически неразрешимых проблем. Кодировка задачи. Длина входа. Понятие сложности алгоритма.

1.2.1 Машины с произвольным доступом (RAM)

В предыдущих лекциях мы довольствовались качественным, интуитивным понятием "эффективного" алгоритма. Для построения же математической теории сложности алгоритмов, разумеется, необходимо строгое количественное определение меры эффективности.

Опыт, накопленный в теории сложности вычислений, свидетельствует, что наиболее удобным и адекватным способом сравнения эффективности разнородных алгоритмов является понятие *асимптотической сложности*, рассмотрению которого и посвящен настоящий параграф.

Первое, о чем следует договориться — это выбор вычислительной модели, в которой конструируются наши алгоритмы. Довольно удивительным образом оказывается, что как раз этот вопрос не имеет слишком принципиального значения для теории сложности вычислений, и тот уровень строгости, на котором мы работали в предыдущем параграфе (число выполнений операторов в алголоподобной записи) оказывается почти приемлемым. Главная причина такого легкомысленного отношения к выбору модели состоит в том, что существуют весьма эффективные способы моделирования (или *трансляции программ* в более привычных терминах) одних естественных вычислительных моделей с помощью других. При этих моделированиях сохраняется класс эффективных алгоритмов и, более того, "как правило", алгоритмы "более эффективные" в одних моделях оказываются более эффективными и в других.

Итак, для начала мы опишем, как следует видоизменить наши алголоподобные алгоритмы, чтобы получить наиболее часто встречающиеся в литературе *машины*⁸ со случайным доступом (*random access machines, RAM*):

- входные данные, результаты промежуточных вычислений и выходное значение хранятся в одномерном массиве $R_1, R_2, \dots, R_n, \dots$ типа **integer**;
- условные операторы и операторы перехода разрешаются лишь в форме
 - **goto** метка,
 - **if условие then goto** метка,

где **условие** должно иметь один из видов $R_i = 0$, $R_i \geq 0$ или $R_i \leq 0$;

- операторы присваивания должны иметь вид $R_i \leftarrow R_j$, $R_i \leftarrow R_j + R_k$, либо $R_i \leftarrow R_j - R_k$, где вместо R_j, R_k могут выступать фиксированные целые числа;
- разрешаются операторы *непрямой адресации* $R_i \leftarrow R_{R_j}$ (на самом деле это характеристическая черта машин со случайным доступом). Иными словами, в

⁸ в теории сложности вычислений под *машинами* традиционно понимают single-purpose machines, т.е. машины, созданные для решения какой-либо одной фиксированной задачи. В привычных терминах это скорее программы.

i -ю ячейку массива можно записать элемент, хранящийся в ячейке с номером R_j .

Мы уже неоднократно использовали (и собираемся свободно использовать в дальнейшем) **for-do-endfor** конструкцию; она легко моделируется на машинах со случайным доступом следующим, хорошо известным образом:

$$\begin{array}{|l} \text{for } i = 1..n \text{ do} \\ \quad \text{тело цикла}(i) \\ \text{endfor} \end{array} \quad \Rightarrow \quad \begin{array}{|l} R \leftarrow 1 \\ \text{label } \ell \\ \text{тело цикла}(R) \\ R \leftarrow R + 1 \\ \text{if } R < n \text{ goto } \ell \end{array} \quad (1.4)$$

где ячейка R и метка ℓ не используются в других частях программы (отметим, что если i встречается в теле цикла в качестве адреса, то в ходе такой трансляции появляются новые операторы не прямой адресации, и это типичный пример их использования). Логические переменные моделируются с помощью целых чисел 0,1 и т.д.

Гораздо более интересен вопрос о том, почему мы не включаем в число основных операций умножение и деление (хотя в литературе такой вариант машин со случайным доступом иногда рассматривается).

Чтобы ответить на этот вопрос, нам следует слегка забежать вперед и решить, чему считать равным *время работы* какой-либо программы при некоторых *фиксированных* исходных данных. С одним возможным определением мы уже познакомились в разделе 1.1 — это число *выполнений* индивидуальных операторов в нашей программе при ее работе на рассматриваемых входных данных. Меры сложности, основанные на таком подходе называются *однородными*. Альтернативная возможность состоит в том, чтобы попытаться также учесть *битовый* размер данных, с которыми оперирует алгоритм. Более точно, назовем временем выполнения индивидуального оператора максимальное *число битов* (т.е. *двоичный логарифм* абсолютного значения) в его аргументах, и определим общее время работы программы на рассматриваемых входных данных как суммарное время выполнения всех индивидуальных операторов. Такие меры сложности называются *логарифмическими*.

Очевидно, логарифмическая сложность всегда больше однородной. С другой стороны, допустим, что однородное время работы некоторой машины со случайным доступом *без умножения и деления* при работе на входных данных R_1, \dots, R_m равно t . Пусть n — максимальная битовая длина чисел R_1, \dots, R_m .

Так как при выполнении любого индивидуального оператора максимальная битовая длина может возрасти не более, чем на 1, в ходе выполнения всей программы встречаются лишь числа битовой длины не более $(t + n)$, и, стало быть, логарифмическая сложность превышает однородную не более, чем в $(t + n)$ раз. В частности (см. обсуждение в разделе 1.1), с точки зрения эффективности, однородная и логарифмическая сложности равносильны, и выбор одной из них в основном определяется внутренней спецификой рассматриваемой задачи. Если последняя имеет

комбинаторно-логическую структуру и не содержит большого количества числовых параметров (входной массив R_1, \dots, R_m состоит в основном из логических переменных), то более удобной и естественной оказывается логарифмическая сложность. Для задач дискретной оптимизации (как правило, содержащих много числовых параметров) преимущественно используется однородная сложность, и в наших оценках мы также будем стараться придерживаться именно этой меры.

Возвращаясь к вопросу об умножении и делении, отметим, что равносильность однородной и логарифмической сложности в присутствии этих операций места уже не имеет. Следующий известный способ быстро переполнить память карманного калькулятора (алгоритм 7)

Алгоритм 7 Быстрое переполнение памяти в присутствии умножения.

Вход: Натуральное t

```

 $R \leftarrow 2$ 
for all  $i \in 1..t$  do
     $R \leftarrow R \times R$ 
end for
```

имеет однородную сложность $(t + 1)$ и логарифмическую порядка экспоненты 2^t . Как показывает обсуждение в разделе 1.1, этот алгоритм не может считаться эффективным с точки зрения логарифмической сложности (хотя и по совершенно другим причинам, нежели переборные алгоритмы), и чтобы избежать неприятных эффектов такого рода мы не включаем умножение (и тем более деление) в список основных операций. В тех же случаях, когда умножение используется "в мирных целях" (под которыми мы в данном случае имеем в виду не гражданские задачи, а задачи более осмысленные, нежели преднамеренное переполнение памяти), его в большинстве случаев можно промоделировать с помощью сложения следующим очевидным образом:

Алгоритм 8 Простой способ вычисления произведения $R_1 \cdot R_2$ на RAM.

Вход: Натуральные R_1, R_2

Выход: $R_1 \times R_2$

```

 $R \leftarrow 0$ 
for all  $i \in 1..R_1$  do
     $R \leftarrow R + R_2$ 
end for
return  $R$ 
```

Отметим, что единственную серьезную конкуренцию машинам со случайным доступом как основы для построения теории сложности вычислений на сегодняшний день составляют так называемые *машины Тьюринга*, которых мы будем рассматривать в следующем разделе. Их принципиальное отличие состоит в отсутствии не прямой индексации: после проведения каких-либо действий с некоторой ячейкой процессор ("головка") может перейти лишь в одну из "соседних" ячеек. Машины Тьюринга, как и логарифмическая сложность, удобны при рассмотрении комби-

наторных задач с небольшим количеством числовых параметров, а также (в силу своей структурной простоты) для теоретических исследований.

1.2.2 Машины Тьюринга и вычислимость

Неформально, *Машина Тьюринга* (далее **МТ**) представляет собой автомат с конечным числом состояний и неограниченной памятью, представленной набором одной или более *лент*, бесконечных в обоих направлениях. Ленты поделены на соответственно бесконечное число ячеек, и на каждой ленте выделена стартовая (нулевая) ячейка. В каждой ячейке может быть записан только один символ из некоторого конечного алфавита Σ , где предусмотрен символ \star для обозначения пустой ячейки.

На каждой ленте имеется головка чтения-записи, и все они подсоединены к "управляющему модулю" МТ — автомату с конечным множеством состояний Γ . Имеется выделенное стартовое состояние "START" и состояние завершения "STOP". Перед запуском МТ находится в состоянии "START", а все головки отпозиционированы на нулевые ячейки соответствующих лент. На каждом шаге все головки считывают информацию из своих текущих ячеек и посылают ее управляющему модулю МТ. В зависимости от этих символов и собственного состояния управляющий модуль производит следующие операции (см. также Рис. 1.3):

1. Посылает каждой головке символ для записи в текущую ячейку каждой ленты
2. Посылает каждой головке одну из команд "LEFT", "RIGHT", "STAY";
3. Выполняет переход в новое состояние (которое, впрочем, может совпадать с предыдущим).

Теперь то же самое более формально.

Машина Тьюринга это набор $T = \langle k, \Sigma, \Gamma, \alpha, \beta, \gamma \rangle$, где

- $k \geq 1$ — натуральное число,
- Σ, Γ — конечные множества входного алфавита и состояний соответственно,
- $\star \in \Sigma$ — символ-пробел, $START, STOP \in \Gamma$ — выделенные состояния,
- α, β, γ — произвольные отображения:

$$\begin{aligned}\alpha &: \Gamma \times \Sigma^k \rightarrow \Gamma \\ \beta &: \Gamma \times \Sigma^k \rightarrow \Sigma^k \\ \gamma &: \Gamma \times \Sigma^k \rightarrow \{-1, 0, 1\}^k\end{aligned}$$

Т.е. α задает новое состояние, β — символы для записи на ленты, γ — как двигать головки. Удобно считать, что алфавит Σ содержит кроме "пробела" \star два выделенных символа, 0 и 1⁹.

⁹Обычно вовсе ограничиваются $\Sigma \equiv \{\star, 0, 1\}$

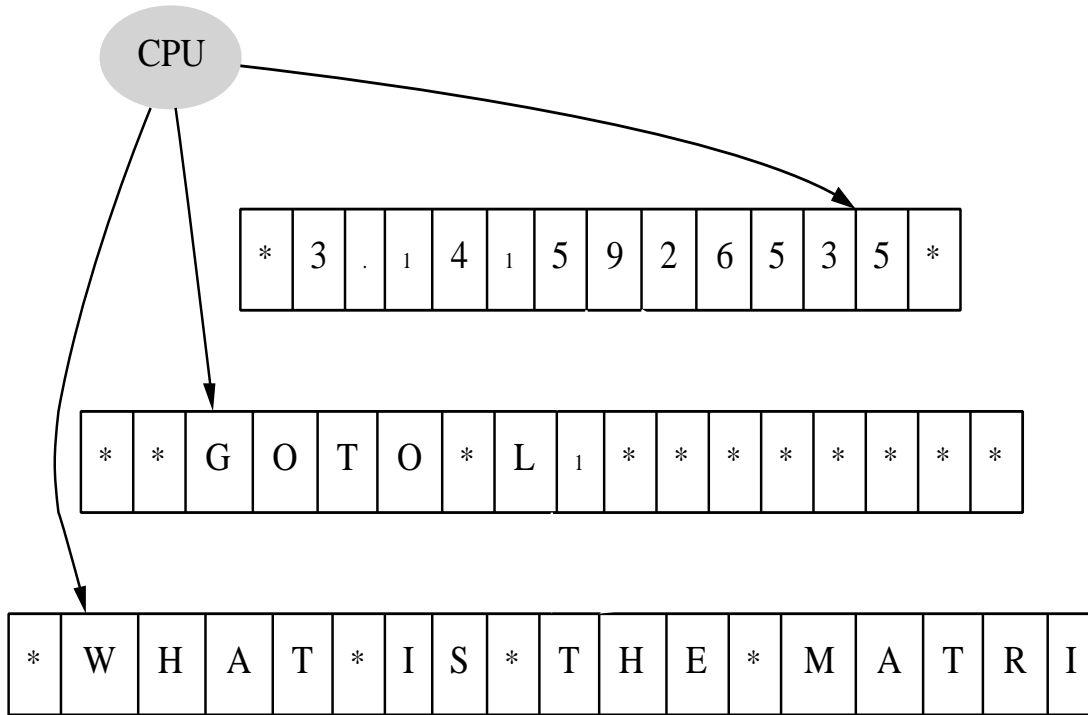


Рис. 1.3: Трехленточная МТ в действии

Под *входом* для МТ подразумевается набор из k слов (k -кортеж слов из Σ^*), записанных на k лентах начиная с нулевых позиций. Обычно, на входные данные записывают только на первую ленту, и под входом x подразумевают k -кортеж $\langle x, 0, \dots, 0 \rangle$.

Результатом работы МТ на некоем входе является также k -кортеж слов из Σ^* , оставшихся на лентах. Для простоты также удобно считать, что *результатом* является только слово на последней ленте, а все остальное — просто мусор. Также считается, что входное слово не содержит пробелов \star — действительно, иначе было бы невозможно определить, где кончается входное слово¹⁰. Алфавит входного слова будем обозначать $\Sigma_0 = \Sigma \setminus \{\star\}$.

Упражнение 4 Постройте машину Тьюринга, которая записывает входное двоичное слово в обратном порядке.

Упражнение 5 Постройте машину Тьюринга, которая складывает два числа, записанные в двоичной системе. Для определенности считайте, что записи чисел разделены специальным символом алфавита "+".

Заметим, что в различной литературе встречается великое множество разновидностей определений МТ — ограничивается число лент, например до одной, причем,

¹⁰Можно конечно разрешить пробелы, \star , но тогда придется зарезервировать еще один символ — "конец ввода".

бесконечной только в одном направлении, или вводится "защита от записи" на все ленты, кроме одной, разумеется, и т.п.

Однако несложно, хотя и несколько утомительно, показать эквивалентность всех этих определений, в смысле вычислительной мощности. Мы тоже слегка затронем эту тему.

Для начала введем понятие *универсальной машины Тьюринга*, которая уже напоминает больше современный программируемый компьютер, чем механическую шкатулку.

Пусть $T = \langle k + 1, \Sigma, \Gamma_T, \alpha_T, \beta_T, \gamma_T \rangle$ и $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$, $k \geq 1$, две МТ, а $p \in \Sigma_0^*$. Итак, T с программой p *симулирует* S , если для произвольного слова $x_1, \dots, x_k \in \Sigma_0^*$,

1. T останавливается на входе (x_1, \dots, x_k, p) тогда и только тогда, если S останавливается на (x_1, \dots, x_k) ;
2. В момент остановки T , на первых k лентах такое же содержимое, как на лентах S , после остановки на том же входе.

Определение 1 $k+1$ ленточная МТ T **универсальна** (для k -ленточных машин), если для любой k -ленточной МТ S (над алфавитом Σ), существует программа $p \in \Sigma_0^*$, на которой T симулирует S .

Теорема 1 Для любого $k \geq 1$ и любого алфавита Σ , существует $(k+1)$ ленточная универсальная МТ.

Доказательство Приведем конструктивное построение универсальной МТ. Основная идея заключается, в том, чтобы разместить на дополнительные ленты универсальной МТ описание моделируемой МТ. Также на дополнительные ленты нужно записывать текущее состояние моделируемой машины S .

Для начала опишем построение с $k + 2$ лентами. Для простоты будем считать, что алфавит Σ содержит символы "0", "1" и "−1". Пусть $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$ произвольная k -ленточная машина Тьюринга. Будем кодировать каждое состояние машины S , словом фиксированной длины r над алфавитом Σ_0^* . Тогда каждую строку из табличного представления машины S можно записать строкой-кодом фиксированной длины:

$$gt_1 \dots t_k \alpha_S(g, t_1, \dots, t_k) \beta_S(g, t_1, \dots, t_k) \gamma_S(g, t_1, \dots, t_k), \quad g \in \Gamma; \quad t_i \in \Sigma; \quad \forall i = 1, \dots, k.$$

Таким образом, на $k + 1$ ленте у нас будет записано все табличное представление машины S , в виде фиксированного размера кодов, на $k + 2$ ленте изначально будет записано стартовое состояние S , на первой ленте — входные данные.

Наша УМТ T будет пробегать по $k + 2$ ленте, пока не совпадут с записанным на $k + 2$ ленте кодом текущее состояние моделируемой машины S и символы $t_1 \dots t_k$

на лентах $1 \dots k$. Тогда из кода извлекаются инструкции, что делать с первыми k лентами, новое состояние, которое записывается на $k + 2$ ленту, и все повторяется.

Для наглядности на Рис. 1.4, приведен граф переходов для 3-х ленточной УМТ, эмулирующей одноленточную МТ. Большие боксы обозначают состояния, вложенные боксы — условия переходов в другие состояния, на ребрах-переходах прописаны совершаемые с лентами действия. $T(k)$, $k = 1, 2, 3$ — обозначают ленты, в контексте сравнения — символ под головкой данной ленты.

Изначально, машина находится в состоянии $START$, на ленте $T(3)$ записан код стартового состояния S . В состоянии $START$ мы пытаемся сравнить текущий код на ленте 2 и текущее состояние S , записанное на ленте 3. Если они совпадают, и совпадает с ожидаемым символ на первой ленте, то "перематываем" к началу третью ленту ($REWIND_T3$), записываем на нее новое состояние из второй ленты ($WRITE_STATE$), записываем на первую ленту символ из второй ленты, двигаем куда надо головку первой ленты ($MOVE$), "перематываем" к началу первую и вторую ленту ($REWIND_T2_T3$), и возвращаемся в исходное состояние. Иначе, по цепочке

$$SKIP_T2_STATE \rightarrow SKIP_T2_TRANSITION \rightarrow SKIP_T2_MOVE$$

(или более короткой) переходим к следующему коду (строке моделируемой МТ) на второй ленте.

Переход от $k + 2$ лент к $k + 1$ несложен — например, достаточно хранить содержимое $k + 2$ ленты в отрицательной области $k + 1$ ленты и моделировать 2 головки на $k + 1$ ленте (одна из которых будет работать с состоянием S , а другая с программой S) методом, описанным в доказательстве следующей теоремы. \square

Следующая теорема утверждает, что в некотором смысле неважно, сколько лент в определении МТ.

Теорема 2 *Для любой k -ленточной МТ S существует одноленточная МТ T , такая, что для любого $x \in \Sigma_0^*$, T останавливается на x , тогда и только тогда, когда на x останавливается S , причем на ленте T записано то, что записано после остановки на последней ленте S . Для разрешимого S за N шагов входа время работы машины T будет $O(N^2)$.*

Доказательство Первым делом мы осуществляем "упаковку" всех лент моделируемой машины S на одну ленту машины T . Мы добиваемся соответствия i -й ячейки j -й ленты моделируемой машины S четной $2(ki + j - 1)$ ячейке единственной ленты машины T . Вернее "упаковывается-растягивается" только входная, первая лента машины S , т.к. остальные ленты S по определению пусты перед запуском. Позиции соответствующие всем лентам S кроме первой, заполняются пробелами. Нечетные позиции $2(ki + j - 1) + 1$ на ленте мы используем для хранения информации о головках машины S — если у машины S в некотором состоянии в i -й позиции j -й ленты стояла головка, то в $2(ki + j - 1) + 1$ ячейку мы запишем 1, иначе пробел \star . Также пометим 0 первые четные ячейки, соответствующие концам лент моделируемой машины S .



Теперь рассмотрим, как T непосредственно моделирует S . Во-первых, T "помнит" (за счет своих собственных состояний, а не дополнительных лент), в каком состоянии должна находиться моделируемая машина S . Также T помнит, какую "ленту" она в данный момент читает. За один проход по своей ленте, T "выясняет" какие символы видны под каждой головкой моделируемой машины S , в какое состояние нужно перевести S , куда нужно двигать головки каждой ленты, и что записывать на каждую ленту. Следующим проходом соответственно двигаются маркеры головок, пишутся символы на моделируемые ленты S .

После окончания моделирования вычисления S , получившийся результат, должен быть "сжат", что аналогично начальному "растяжению".

Очевидно, что описанная таким образом машина T , вычисляет тоже, что и моделируемая машина S . Теперь оценим число шагов T . Пусть M , число сканированных машиной T ячеек. Очевидно, что $M = O(N)$ (кстати, почему?). Моделирования каждого шага S требует $O(M)$ шагов, таким образом, все моделирование состоит из $O(MN)$ шагов. Начальное "растяжение" и конечное "сжатие" требуют по $O(M^2)$ шагов.

Итак, мы получаем, что все моделирование требует не более $O(N^2)$ шагов. \square

Таким образом, при рассуждениях можно ограничиваться рассмотрением одноленточных МТ, причем подразумевать под МТ ее программу и использовать выражения вроде "подать на вход машины Тьюринга T_1 машину Тьюринга T_2 ".

Теперь уже можно ввести строгое определение вычислимости.

Определение 2 Функция $f : N \rightarrow N$ является **вычислимой**, если существует такая машина Тьюринга \mathbf{T} , что если на вход ей подать x (представленным в некоторой кодировке), то

1. если функция f определена на x , и $f(x) = y$, то машина \mathbf{T} останавливается на входе x , и на выходе у нее записано y .
2. если функция f не определена на x , то машина \mathbf{T} заклинивается (не останавливается за любое конечное число шагов) на входе x .

Аналогичным образом определяется понятие разрешимости и вычислимости для языков и других множеств.

Определение 3 Множество S (язык L) является **разрешимым**, если существует такая машина Тьюринга \mathbf{T} , что если на вход ей подать элемент $x \in S$ (слово $l \in L$), то она остановится и выведет 1. Иначе ($x \notin S$, $l \notin L$), T останавливается и выводит 0.

Сразу возникает вопрос, любую ли функцию $y = f(x)$ ¹¹, можно вычислить на МТ?

¹¹Можно подразумевать функции на множестве натуральных чисел, или преобразования строк — очевидно, что это равнозначно.

Интересно, что до 1930 года все математики были уверены в разрешимости любой корректно сформулированной математической задачи. Однако эта вера была разрушена в 1931 году австрийским математиком Куртом Геделем.

В алгоритмическом смысле, вопрос о разрешимости некоторого множества или языка, означает вопрос о существовании алгоритма, разрешающего это множество (язык).

В 1936 году Черч формализовал понятие алгоритма через так называемые *рекурсивные функции*, — понятия из математической логики (см. например [E.M84]) и доказал существование алгоритмически неразрешимых задач.

Понятие алгоритма и вычислимости было формализовано также Тьюрингом, через рассмотренное нами определение машины Тьюринга. Оказалось, что модели Черча и Тьюринга эквивалентны и определяют один и тот же класс вычислимых функций. Более того, со времени первого определения понятия алгоритма было предложено множество различных моделей вычислений, зачастую весьма далеких от машин Тьюринга, RAM или даже реальных ЭВМ, однако никому еще не удалось доказать, что существует хоть одна задача, не вычислимая по Тьюрингу, но вычислимая в рамках другой парадигмы вычислений.

Этот факт сегодня выражается общей верой в гипотезу, называемую *тезисом Черча*, что любая функция, вычислимая на любом вычислительном устройстве, вычислима и на машине Тьюринга. Таким образом, если мы принимаем эту гипотезу, то можем смело говорить о вычислимости, не указывая конкретную модель вычислений.

Как проще всего убедиться в существовании невычислимых функций? Ответом служит формулировка следующего упражнения.

Упражнение 6 *Докажите, что существуют невычислимые по Тьюрингу, функции $y = f(x)$. Использовать мощностные соображения.*

Указание: рассмотреть мощность множества машин Тьюринга и мощность множества булевых функций $f : N \rightarrow \{0, 1\}$

Несмотря на то, что ответ получен в предыдущем упражнении, он несколько неконструктивный, и не дает возможности "познакомиться" с представителем неразрешимой задачи (невычислимой функции).

Рассмотрим классическую неразрешимую задачу.

Задача 5 Проблема останова (halting problem). *Для данной машины Тьюринга M и входа x определить, остановится ли машина Тьюринга M , начиная работу на x ?*

Теорема 3 *Проблема останова алгоритмически неразрешима.*

Доказательство От противного. Предположим, что есть такой алгоритм, т.е. существует машина Тьюринга T , которая на входе (M, x) ¹² дает ответ "да", если машина M останавливается на входе x , в противном случае дает ответ "нет". Тогда есть и такая машина $T^{diag}(X) \equiv T(X, X)$, которая на входе X моделирует работу T на "диагональном" входе (X, X) .

Надстроим над $T^{diag}(X)$ машину $T^{fraud}(X)$, которая если ответ машины T^{diag} — "да", то T^{fraud} начинает двигать головку вправо и не останавливается (зацикливается), а если ответ T^{diag} — "нет", то T^{fraud} останавливается.

Остановится ли T^{fraud} на входе T^{fraud} ? 1) Если да, то T^{diag} дает ответ "нет" на входе T^{fraud} , т.е. утверждает, что T^{fraud} не должна останавливаться на T^{fraud} .

2) Если не остановится, то T^{diag} дает ответ "да" на входе T^{fraud} , т.е. утверждает, что T^{fraud} должна останавливаться на T^{fraud} . Противоречие. \square

Упражнение 7 Докажите, что также неразрешима версия задачи 5, остановка на пустом слове. А именно, для данной МТ T , определить, остановится ли она на пустом слове.

Упражнение 8 Докажите, что неразрешима Проблема недостижимого кода — существует ли алгоритм, который для заданной машины Тьюринга T , и ее состояния q_k отвечает на вопрос, попадет ли машина в это состояние хотя бы для одного входного слова x (или это состояние недостижимо).

Упражнение 9 Докажите, что не существует алгоритма, который выписывает одну за другой все машины Тьюринга, которые не останавливаются, будучи запущенными на пустой ленте.

Упражнение 10 Существует ли алгоритм, который выписывает одну за другой все машины Тьюринга, которые останавливаются, будучи запущенными на пустой ленте?

1.3 Сложность алгоритмов

1.3.1 Сложность в худшем случае (Worst Case Complexity)

В предыдущем разделе 1.2 мы формально определили понятие алгоритма, вычислимости, и ввели несколько эквивалентных моделей вычислений.

При этом, нас совершенно не интересовало потребление вычислительных ресурсов, необходимое любому алгоритму для какой-либо вычислимой функции.

В рамках введенных моделей последовательных вычислений, основными вычислительными ресурсами являются *время*, затраченное алгоритмом на вычисление, и

¹²Под машиной Тьюринга M на входе подразумевается ее описание.

использованная *память*. Понятно, что потребление этих ресурсов может зависеть от размера входной задачи.

Значит, сначала надо определить, что понимать под размером входных данных. По аналогии с однородными и логарифмическими мерами сложности возникают две различные возможности. Мы можем определить размер как *размерность* задачи, т.е. число ячеек RAM, занятых входными данными. Например, размерность задач КОММИВОЯЖЕР, КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ, рассмотренных в разделе 1.1, равна $\frac{m(m-1)}{2}$. Либо мы можем определить размер входных данных как суммарную *битовую длину* записи всех входных параметров ($\sum_{1 \leq i < j \leq m} \lceil \log_2(|d_{ij}| + 1) \rceil$ для вышеупомянутых задач)—число ячеек на входной ленте МТ.

Все алгоритмы, рассмотренные нами в разделе 1.1, обладают тем приятным свойством, что время их работы относительно однородной меры сложности зависит лишь от размерности входных данных и не зависит от их значений. Более того, тем же свойством будет обладать любой алгоритм, который содержит все циклы в котором имеют вид:

for $i = 1$ **to** *ограничитель* **do**,

где *ограничитель* — некоторая явная функция, зависящая только от размерности задачи. Мы будем называть алгоритмы с этим свойством *однородными*. *Сложность* однородного алгоритма — это целочисленная функция целого аргумента $t(n)$, равная однородному времени его работы для входных данных размерности n .

К сожалению, эффективность и однородность — требования, зачастую плохо совместимые между собой, и с интересными примерами такого рода мы еще познакомимся¹³. Поэтому нам необходимо иметь меру сложности, пригодную также для неоднородных алгоритмов.

Естественная попытка определить сложность неоднородного алгоритма как функцию $t(R_1, \dots, R_m)$ от всего массива исходных данных при ближайшем рассмотрении оказывается неудовлетворительной: эта функция несет в себе огромное количество избыточной информации, с трудом поддается вычислению или хотя бы оцениванию в явном виде и позволяет сравнивать между собой различные алгоритмы лишь в исключительных случаях. Поэтому, чтобы получить на самом деле работающее на практике определение сложности, нам необходимо каким-нибудь образом свести эту функцию к функции *одного* (или в крайнем случае небольшого числа) целого аргумента, как в случае однородных алгоритмов. Имеются два принципиально различных подхода к решению этой задачи.

Самый распространенный подход — это рассмотрение *сложности в наихудшем случае*. Мы полагаем $t(n)$ равным

$$\max \left\{ t(R_1, \dots, R_m) \mid \sum_{i=1}^m \lceil \log_2(|R_i| + 1) \rceil \leq n \right\}.$$

Иными словами, $t(n)$ — это то время работы, которое данный алгоритм может *гарантировать заведомо*, если известно, что суммарная битовая длина входных

¹³ Яркий пример — симплекс-метод для линейного программирования.

данных не превышает n . При этом время работы алгоритма для некоторых (или даже для "большинства") таких входных данных может быть существенно меньше $t(n)$.

Более формально,

Определение 4 Пусть $t : N \rightarrow N$. Машина Тьюринга T имеет временную сложность¹⁴ $t(n)$, если для каждого входного слова длины n T выполняет не больше $t(n)$ шагов до остановки. Также будем обозначать временную сложность машины Тьюринга T , как $\text{time}_T(n)$.

Другая возможность заключается в рассмотрении *сложности в среднем*. В этой постановке вводится некоторое (например, равномерное) вероятностное распределение на массивах входных данных битовой длины $\leq n$, и сложностью $t(n)$ называется *математическое ожидание* времени работы нашего алгоритма на случайном входе, выбранном в соответствии с этим распределением. Несмотря на свою внешнюю привлекательность, сложность в среднем в настоящее время не может конкурировать со сложностью в наихудшем случае по объему проводимых исследований. Причины такого отставания вполне прагматичны: к настоящему времени не удалось выработать единого мнения о том, что такое "естественное" распределение на массивах входных данных для классических задач, которое реально встречалось бы в большей (или по крайней мере в большой) доле их практических применений.

1.3.2 Полиномиальные алгоритмы

По каким же критериям следует различать разнородные алгоритмы для одной и той же задачи? — по *асимптотическому* поведению их сложности $t(n)$. В теории сложности вычислений традиционно игнорируются мультипликативные константы в оценках сложности. Вызвано это тем, что в большинстве случаев эти константы приносятся некоторыми малоинтересными внешними факторами, и их игнорирование позволяет абстрагироваться от такого влияния. Приведем несколько примеров таких факторов:

- при несущественных модификациях вычислительной модели сложность обычно изменяется не более, чем на некоторую (как правило, довольно небольшую) мультипликативную константу. Например, если мы разрешим в машинах со случайным доступом **for-do-endfor** конструкцию, то, ввиду наличия моделирования (1.4), сложность от этого может уменьшиться не более, чем в два раза. Следовательно, если мы игнорируем мультипликативные константы, мы можем свободно использовать в машинах со случайным доступом **for-do-endfor** циклы и другие алголоподобные конструкции, не оговаривая всякий раз этого специально.

¹⁴time complexity

- При переходе от двоичной к восьмеричной или десятичной системе счисления длина битовой записи, а, следовательно, и сложность полиномиальных алгоритмов изменяется не более, чем на мультипликативную константу. Таким образом, путем игнорирования таких констант мы также можем абстрагироваться от вопроса о том, в какой системе счисления мы работаем и вообще не указывать явно в наших оценках основание логарифмов.
- Существует

Теорема 4 *Теорема о линейном ускорении (Linear Speedup Theorem). Для любой машины Тьюринга T , определяющей язык L , и произвольной константы $c \geq 0$, существует машина Тьюринга S^{15} , над тем же алфавитом, разрешающая тот же язык L , причем $time_S(n) \leq c \cdot time_T(n) + n$.*

- Имеет место приблизительное равенство

$$\begin{aligned} &\text{физическое время работы} = \\ &\text{физическое время выполнения одного оператора} \times \\ &\text{однородная сложность.} \end{aligned}$$

В этой формуле теория сложности вычислений отвечает за второй сомножитель, а первый зависит от того, насколько грамотно составлена программа, от качества транслятора на язык низшего уровня (что определяет, например, сколько времени занимает доступ к RAM или твердому диску), быстродействия вычислительной техники и т.д. Несмотря на безусловную важность этих вопросов, они выходят за рамки теории сложности вычислений, и игнорирование мультипликативных констант позволяет от них абстрагироваться и выделить интересующий нас второй сомножитель в чистом виде.

По этой причине в теории сложности вычислений широкое распространение получили "О-большое" обозначения. Типичный результат выглядит следующим образом: "данный алгоритм работает за время $O(n^2 \log n)$ ", и его следует понимать как "существует такая константа $c > 0$, что время работы алгоритма в наихудшем случае не превышает $cn^2 \log n$ ". Практическая ценность асимптотических результатов такого рода зависит от того, насколько мала неявно подразумеваемая константа c . Как мы уже отмечали выше, для подавляющего большинства известных алгоритмов она находится в разумных пределах, поэтому, как правило, имеет место следующий тезис: *алгоритмы, более эффективные с точки зрения их асимптотического поведения, оказываются также более эффективными и при тех сравнительно небольших размерах входных данных, для которых они реально используются на практике* (одним из наиболее заметных исключений из этого правила является задача матричного умножения — см. [SS71]).

Таким образом, теория сложности вычислений *по определению* считает, что алгоритм, работающий за время $O(n^2 \log n)$ лучше алгоритма с временем работы

¹⁵ доказательство конструктивно — машину S можно построить по машине T .

$O(n^3)$, и в подавляющем большинстве случаев это отражает реально существующую *на практике* ситуацию. И такой способ сравнения эффективности различных алгоритмов оказывается достаточно универсальным — сложность в наихудшем случае с точностью до мультипликативной константы для подавляющего большинства реально возникающих алгоритмов оказывается достаточно гладкой и простой функцией, и практически всегда функции сложности разных алгоритмов можно сравнить между собой по их асимптотическому поведению¹⁶.

Таким образом, можно определять классы алгоритмов, замкнутых относительно выбора вычислительной модели. Например,

Определение 5 Алгоритм называется **полиномиальным**, если его сложность в наихудшем случае $t(n)$ ограничена сверху некоторым полиномом (многочленом) от n .

Это определение оказывается вполне независимым относительно выбора вычислительной модели: любые "разумные" модели (более-менее любые модели, в которых не может происходить экспоненциального нарастания памяти, как в алгоритме 7) оказываются равносильными, с его точки зрения. Если для данного алгоритма ограничена полиномом его сложность в среднем (относительно некоторого распределения на входных данных), то и сам алгоритм называется *полиномиальным в среднем* (относительно того же распределения).

Полиномиальные алгоритмы противопоставляются *экспоненциальным*, т.е. таким, для которых $t(n) \geq 2^{n^\epsilon}$ для некоторой фиксированной константы $\epsilon > 0$ и почти всех n . Экспоненциальными являются алгоритмы 4, 7 (относительно однородной меры сложности) и 8.

В теории сложности задач дискретной оптимизации естественным образом возникают по крайней мере две модификации этого понятия, одно из которых является его усилением, а другое — ослаблением.

Предположим, что полиномиальный алгоритм к тому же еще и однороден. Тогда он обладает следующими двумя свойствами (второе из которых, как мы видели выше, влечет первое):

1. Длина битовой записи всех чисел, возникающих в процессе работы алгоритма ограничена полиномом от длины битовой записи входных данных (это свойство на самом деле присуще любому полиномиальному алгоритму).
2. Число арифметических операций (однородная сложность) ограничено полиномом от *размерности* задачи.

¹⁶следует оговориться, что иногда рассматривают функции сложности, зависящие от двух-трех параметров, например, от числа вершин и ребер входного графа, или вертикального и горизонтального размера входной матрицы. Тем не менее, даже в таких случаях сравнение асимптотической сложности различных алгоритмов все равно удается провести довольно часто.

Основная причина, по которой мы предпочли не рассматривать умножение и деление в качестве основных операций, состоит в том, что это нарушило бы импликацию $2 \Rightarrow 1$ (алгоритм 7). Если мы тем не менее разрешим умножение и деление и при этом потребуем, чтобы спорное свойство 1 обеспечивалось внутренней структурой алгоритма, то мы получим класс алгоритмов, называемых в литературе *сильно полиномиальными алгоритмами*. Таким образом, сильно полиномиальные алгоритмы обобщают однородные полиномиальные алгоритмы; хорошим примером сильно полиномиального алгоритма который, по-видимому, нельзя привести к однородному виду (т.е. избавиться от делений и умножений), служит известный алгоритм Гаусса решения систем линейных уравнений.

Итак, отличительной чертой сильно полиномиальных алгоритмов является то, что время их работы ограничено полиномом от *размерности* задачи. Иными словами, "длина" любого входного параметра полагается равной 1, независимо от его фактического значения. На другом полюсе находятся *псевдополиномиальные алгоритмы*, в которых требуется, чтобы время работы алгоритма было полиномиально лишь от суммы *абсолютных значений* (а не от битовой длины записи) входящих в задачу числовых параметров: с типичными примерами таких алгоритмов мы еще познакомимся.

1.3.3 Полиномиальность и эффективность

Может ли полиномиальный алгоритм быть неэффективным? Разумеется может, если в полиномиальной оценке $t(n) \leq C \cdot n^k$ времени его работы либо мультипликативная константа C либо показатель k чрезмерно велики. Опыт показывает, что такое случается крайне редко, и подавляющее большинство полиномиальных алгоритмов для естественных задач удовлетворяет оценке $t(n) \leq 10 \cdot n^3$. В тех же немногих случаях, когда полиномиальный алгоритм оказывается малоприменимым с практической точки зрения, это обстоятельство всегда стимулирует бурные исследования по построению на его основе действительно эффективного алгоритма, что как правило, приводит к интересным самим по себе побочным следствиям. Хрестоматийным примером такого рода может служить метод эллипсоидов для линейного программирования [L.G79].

Может ли неполиномиальный алгоритм быть эффективным? Может, и по меньшей мере по трем причинам. Во-первых, может случиться так, что примеры, на которых время работы алгоритма велико, настолько редки, что вероятность обнаружить хотя бы один из них на практике пренебрежимо мала. В математических терминах это означает, что алгоритм полиномиален в среднем относительно любого "разумного" распределения и, по-видимому, под эту категорию попадает симплекс-метод ([A.91]).

Во-вторых, многие псевдополиномиальные алгоритмы являются эффективными когда возникающие на практике числовые параметры не слишком велики.

Еще раз подчеркнем, что примеров задач, на которых нарушается основополагающее равенство

"полиномиальность"- "эффективность"

крайне мало по сравнению с числом примеров, на которых оно блестяще подтверждается. Поэтому в дальнейшем мы отождествляем эти два понятия (если не оговорено противное), а также рекомендуем читателю вернуться еще раз к разделу 1.1 и убедиться, что все сделанные в нем выводы сохраняют свою силу после замены слова "эффективный" на "полиномиальный".

1.4 Эффективность и классы DTIME, DSPACE

Следующим естественным шагом в выполнении нашей программы могло бы стать разумное определение "оптимального" алгоритма для данной алгоритмической задачи. К сожалению, в рассматриваемом контексте такой подход бесперспективен, и соответствующий результат (*теорема об ускорении*), установленный на заре развития теории сложности вычислений в работе [Blu67] послужил на самом деле мощным толчком для ее дальнейшего развития. Мы приведем этот результат (без доказательства) в ослабленной, но зато весьма красноречивой форме:

Теорема 5 *Существует разрешимая алгоритмическая задача, для которой выполнено следующее. Для произвольного алгоритма, решающего эту задачу и имеющего сложность в наихудшем случае $t(n)$, найдется другой алгоритм (для этой же задачи) со сложностью $t'(n)$ такой, что*

$$t'(n) \leq \log_2 t(n)$$

выполнено для почти всех n (т.е. для всех n , начиная с некоторого).

Иными словами, любой алгоритм, решающий эту задачу, можно существенно ускорить, т.е. отыскать алгоритм намного меньшей асимптотической сложности. Следует сразу отметить, что задача, о которой идет речь в этой теореме, выглядит довольно искусственно, и, по-видимому, ничего подобного не происходит для задач, реально возникающих на практике. Тем не менее, теорема об ускорении не позволяет нам определить *общее* математическое понятие "оптимального" алгоритма, пригодное для всех задач, поэтому развитие теории эффективных алгоритмов пошло другим путем. Именно, одним из центральных понятий этой теории стало понятие *класса сложности*. Так называется совокупность тех алгоритмических задач, для которых существует *хотя бы один* алгоритм с теми или иными сложностными характеристиками. В наших лекциях нам в основном понадобятся следующие классы сложности: **P**, **BPP**, **PSPACE**, **EXPTIME** и **NP**; читателю, интересующемуся более глобальной картиной *сложностной иерархии*, мы рекомендуем обратиться к обзору [Joh90].

Для формальных определений классов сложности обычно рассматривают¹⁷ не произвольные алгоритмы, а алгоритмы для так называемых *задач разрешения*, ко-

¹⁷О причинах будет говориться в следующих разделах.

гда требуется определить принадлежит или нет некоторый элемент множеству. Учитывая необходимость кодирования данных, подаваемых на вход машине Тьюринга, эти задачи абсолютно эквивалентны задачам распознавания языков, когда на некотором алфавите Σ рассматривается подмножество слов $L \subset \Sigma^*$, и для произвольного слова $l \in \Sigma^*$, нужно определить, принадлежит ли оно языку L . Таким образом, каждый язык L определяет одну из задач разрешения, для которых мы сейчас более формально определим классы временной сложности.

Определение 6 Язык $L \subset \Sigma^*$ принадлежит классу $DTIME(t(n))$, если существует машина Тьюринга T , разрешающая данный язык, и $time_T(n) \leq t(n)$.

Определение 7

$$\mathbf{P} = \cup_{c>0} DTIME(n^c)$$

$$\mathbf{EXPTIME} = \cup_{c>0} DTIME(2^{n^c})$$

Иными словами класс P состоит из тех алгоритмических задач, которые допускают решение хотя бы одним полиномиальным (в наихудшем случае) алгоритмом. Например, алгоритм 5 полиномиален, поэтому задача КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ принадлежит классу P . Этому же классу принадлежит и задача МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО.

Задача УМНОЖЕНИЕ двух чисел также лежит в P , хотя алгоритм 8 и неполиномиален (в самом деле, его однородная сложность равна $R_2 + 1$, что не ограничено никаким полиномом от битовой длины $\lceil \log_2(|R_2| + 1) \rceil$). Полиномиальный алгоритм для УМНОЖЕНИЯ легко строится, например, с помощью моделирования обычного умножения столбиком (существуют еще более эффективные нетривиальные алгоритмы).

Обычно, обобщение классов сложности, введенных для задач разрешения, на произвольные вычислимые функции $f : N \rightarrow N$, происходит путем ассоциирования с произвольной вычислимой функцией $y = f(x)$, задачи разрешения "Для данных y, x проверить, правда ли, что $y = f(x)$ ".

В современной теории сложности вычислений понятие полиномиального алгоритма является крайне удачным и адекватным математическим уточнением интуитивного понятия "эффективный алгоритм" (См. далее раздел 1.3.3). Тем самым класс P представляет собой класс эффективно решаемых задач.

Следующими по значимости, после рассмотренных выше временных мер и классов сложности, являются меры сложности, отражающие используемый алгоритмом объем памяти, т.е. (в наихудшем случае) максимальное число ячеек R_i , используемых алгоритмом на входах размера $\leq n$. Более формально,

Определение 8 k -ленточная машина Тьюринга (произвольное $k > 0$) T имеет пространственную сложность $s(n)$, если для любого входного слова длины n , T просматривает не более $s(n)$ ячеек на всех лентах.

Определение 9 Язык $L \subset \Sigma^*$ принадлежит классу $\mathbf{DSPACE}(s(n))$, если существует машина Тьюринга T , разрешающая данный язык, и пространственная сложность T не превосходит $s(n)$.

Определение 10

$$\mathbf{PSPACE} = \cup_{c>0} \mathbf{DSPACE}(n^c)$$

Очевидно, что \mathbf{P} содержится в классе задач \mathbf{PSPACE} , разрешимых с полиномиальной памятью, просто в силу того, что один оператор может вводить не более одной новой ячейки памяти. Обратное, по-видимому, неверно (хотя и не доказано строго — см. обсуждение в разделе 1.6). Наконец, стоит также упомянуть, что существуют различные меры сложности и сложностные классы, связанные с параллельными и распределенными вычислениями.

Упражнение 11 Покажите $\mathbf{P} \subseteq \mathbf{EXPTIME}$.

Упражнение 12 Покажите $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$.

1.5 Схемы и схемная сложность

Этот раздел основан на соответствующей главе [A. 99].

Схема (булева) — это способ вычислить функцию $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$.

Помимо исходных переменных x_1, \dots, x_n , для которых вычисляется значение f , схема использует некоторое количество вспомогательных переменных y_1, \dots, y_s и некоторый набор (базис) булевых функций \mathcal{F} .

Схема S в базисе \mathcal{F} определяется последовательностью присваиваний Y_1, \dots, Y_s .

Каждое присваивание Y_i имеет вид

$$y_i := f_j(u_{k_1}, \dots, u_{k_r}),$$

где $f_j(\cdot) \in \mathcal{F}$, а переменная u_{k_p} ($1 \leq p \leq r$) — это либо одна из исходных переменных x_t ($1 \leq t \leq n$), либо вспомогательная переменная y_l с меньшим номером ($1 \leq l < i$).

Таким образом, для каждого набора значений исходных переменных последовательное выполнение присваиваний, входящих в схему, однозначно определяет значения всех вспомогательных переменных. *Результатом вычисления* считаются значения последних m переменных y_{s-m+1}, \dots, y_s .

Схема *вычисляет* функцию f , если для любых значений x_1, \dots, x_n результатом вычисления является $f(x_1, \dots, x_n)$.

Определение 11 Схема называется **формулой**, если каждая вспомогательная переменная используется в правой части присваиваний только один раз.

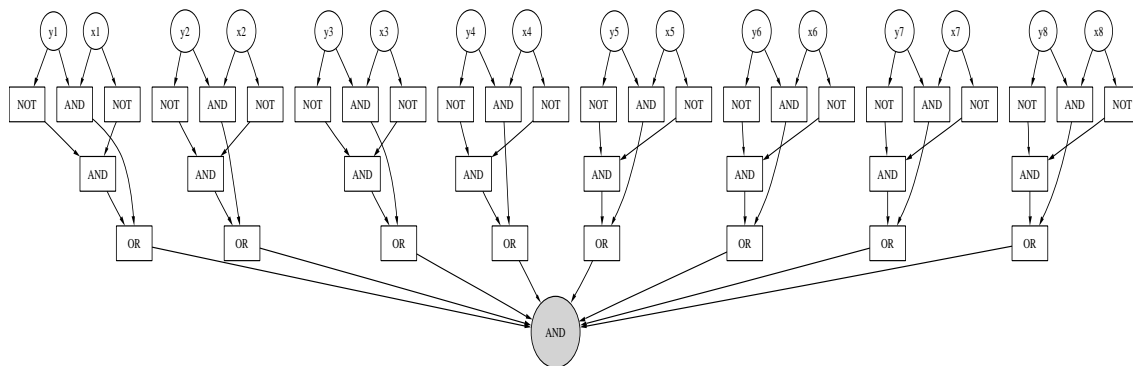


Рис. 1.5: Схема сравнения двух битовых строк

Обычные математические формулы именно так задают последовательность присваиваний: "внутри" формул не принято использовать ссылки на их части или другие формулы.

Схему можно также представлять в виде ориентированного ациклического (См. Рис. 1.5) графа, у которого вершины входной степени 0 (*входы*) помечены исходными переменными; остальные вершины (*функциональные элементы*) помечены функциями из базиса; ребра помечены числами, указывающими номера аргументов; вершины выходной степени 0 (*выходы*) помечены переменными, описывающими результат работы схемы.

Вычисление на графе определяется индуктивно: как только известны значения всех вершин y_1, \dots, y_{k_v} , из которых ведут ребра в данную вершину v , вершина v получает значение $y_v = f_v(y_1, \dots, y_{k_v})$, где f_v — базисная функция, которой помечена вершина.

При переходе к графу схемы мы опускаем *несущественные присваивания*, которые ни разу не используются на пути к выходным вершинам, так что они никак не влияют на результат вычисления.

Определение 12 *Базис называется полным, если для любой булевой функции f есть схема в этом базисе, вычисляющая f .*

Ясно, что в полном базисе можно вычислить произвольную функцию $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$ (такую функцию можно представить как упорядоченный набор из m булевых функций).

Булева функция может быть задана таблицей значений. Приведем таблицы значений для трех функций

$$NOT(x) = \neg x, \quad OR(x_1, x_2) = x_1 \vee x_2, \quad AND(x_1, x_2) = x_1 \wedge x_2$$

(отрицание, дизъюнкция, конъюнкция), образующих полный базис, который будем считать стандартным. В дальнейшем имеются в виду схемы именно в этом базисе,

если явно не указано что-либо иное.

x	NOT	x_1	x_2	OR	x_1	x_2	AND
0	1	0	0	0	0	0	0
1	0	0	1	1	0	1	0
		1	0	1	1	0	0
		1	1	1	1	1	1

Конъюнкция и дизъюнкция определяются для произвольного числа булевых переменных аналогичным образом: конъюнкция равна 1 только тогда, когда все аргументы равны 1, а дизъюнкция равна 0 только тогда, когда все аргументы равны 0. В стандартном базисе они очевидным образом вычисляются схемами (и даже формулами) размера $n - 1$.

Теорема 6 *Базис $\{NOT, OR, AND\}$ — полный.*

Доказательство *Литералом* будем называть переменную или ее отрицание. Конъюнкцией литералов (это схема и даже формула) легко представить функцию $\chi_u(x)$, которая принимает значение 1 ровно один раз: при $x = u$. Если $u_i = 1$, включаем в конъюнкцию переменную x_i , если $u_i = 0$, то включаем в конъюнкцию $\neg x_i$.

Произвольная функция f может быть представлена в виде

$$f(x) = \bigvee_{u: f(u)=1} \chi_u(x). \quad (1.5)$$

В таком случае говорят, что f представлена в *дизъюнктивной нормальной форме* (ДНФ), т.е. как дизъюнкция конъюнкций литералов.¹⁸

Как уже говорилось, дизъюнкция нескольких переменных выражается формулой в стандартном базисе. \square

Определение 13 *Размером схемы называется количество присваиваний в схеме.*

Определение 14 *Глубиной схемы называется максимальное число элементов на пути от входов к выходу.*

Определение 15 *Минимальный размер схемы в базисе \mathcal{F} , вычисляющей функцию f , называется **схемной сложностью** функции f в базисе \mathcal{F} и обозначается $s_{\mathcal{F}}(f)$.*

¹⁸Далее нам еще потребуются и *конъюнктивная нормальная форма* (КНФ) — конъюнкция дизъюнкций литералов.

Переход от одного полного конечного базиса к другому полному конечному базису меняет схемную сложность функций на множитель $O(1)$. Так что в асимптотических оценках выбор конкретного полного базиса неважен и поэтому будем использовать обозначение $c(f)$ для схемной сложности f в конечном полном базисе.

Каждый предикат f на множестве $\{0, 1\}^*$ определяет последовательность булевых функций $f_n: \{0, 1\}^n \rightarrow \{0, 1\}$ следующим образом:

$$f_n(x_1, x_2, \dots, x_n) = f(x_1 x_2 \dots x_n),$$

где справа стоит характеристическая функция предиката f .


Определение 16 Предикат f принадлежит классу $P/poly$, если

$$c(f_n) = \text{poly}(n).$$

Теорема 7 $P \subset P/poly$.

Доказательство Если МТ работает за полиномиальное время, то и память, которую она использует, ограничена полиномом. Поэтому весь процесс вычисления на входном слове x длины n можно представить *таблицей вычисления* размера $T \times S$, где $T = \text{poly}(n)$, $S = \text{poly}(n)$.

$t = 0$		$\Gamma_{0,1}$		
$t = 1$				
	...			
$t = j$		Γ'_{left}	Γ'	Γ'_{right}
$t = j + 1$			Γ	
	...			
$t = T$...			



 S клеток

Строка с номером j таблицы задает состояние МТ после j тактов работы. Символы $\Gamma_{j,k}$, записанные в таблице, принадлежат алфавиту $\mathcal{S} \times \{\emptyset \cup \mathcal{Q}\}$. Символ $\Gamma_{j,k}$ определяет пару (символ, записанный в k -й ячейке после j тактов работы; состояние управляющего устройства после j тактов работы, если головка находится над k -й ячейкой, в противном случае второй элемент пары — \emptyset). Для простоты также считаем, что если вычисление заканчивается при некотором входе за $T' < T$ тактов, то строки с номерами, большими T' , повторяют строку с номером T' .

Построить схему, вычисляющую значения предиката на словах длины n , можно следующим образом. Состояние каждой клетки таблицы можно закодировать конечным (не зависящим от n) числом булевых переменных. Имеются *локальные*

правила согласования, т.е. состояние каждой клетки Γ в строке ниже нулевой однозначно определяется состояниями клеток в предыдущей строке, лежащих непосредственно над данной (Γ'), левее данной (Γ'_{left}) и правее данной (Γ'_{right}). Каждая переменная, кодирующая состояние клетки Γ , есть функция от переменных, кодирующих состояния клеток Γ'_{left} , Γ' , Γ'_{right} . Все эти функции могут быть вычислены схемами конечного размера. Объединяя эти схемы, получим схему, вычисляющую все переменные, кодирующие состояния клеток таблицы; размер этой схемы будет $O(ST) = O(n^{O(1)})$.

Осталось заметить, что переменные, кодирующие часть клеток нулевой строки, определяются входным словом, а переменные, кодирующие остальные клетки нулевой строки, являются константами. Чтобы узнать результат вычисления, нужно определить символ, записанный в нулевой ячейке ленты в конце вычисления.

Без ограничения общности можно считать, что состояния клеток таблицы кодируются так, что одна из кодирующих переменных равна 1 только в том случае, когда в ячейке записана 1. Тогда значение этой переменной для кода $\Gamma_{T,0}$ и будет результатом вычисления. \square

Класс $P/poly$ шире класса P . Любой функции от натурального аргумента $\varphi(n)$ со значениями в $\{0, 1\}$ можно сопоставить предикат f_φ по правилу $f_\varphi(x) = \varphi(|x|)$, где $|x|$ обозначает длину слова x . Ограничение такого предиката на слова длины n тождественно равно 0 или 1 (в зависимости от n). Схемная сложность таких функций ограничена константой. Поэтому все такие предикаты по определению принадлежат $P/poly$, хотя среди них есть и неразрешимые предикаты.

Справедливо следующее усиление теоремы.

Теорема 8 *f принадлежит P тогда и только тогда, когда*

1. $f \in P/poly$;
2. существует МТ, которая по числу n за время $poly(n)$ строит схему вычисления f_n .

Доказательство \Rightarrow Данное в доказательстве теоремы 7 описание нетрудно превратить в МТ, которая строит схему вычисления f_n за полиномиальное по n время (схема f_n имеет простую структуру: каждая переменная связана с предыдущими одними и теми же правилами согласования).

\Leftarrow Столь же просто. Вычисляем размер входного слова. Затем строим по этому размеру схему $S_{|x|}$ вычисления $f_{|x|}$, используя указанную в условии 2) машину. После этого вычисляем $S_{|x|}(x)$ на машине, которая по описанию схемы и значениям входных переменных вычисляет значение схемы за полиномиальное от длины входа время. \square

Упражнение 13 *Постройте полиномиальный алгоритм, определяющий, является ли данный базис полным. Базисные функции заданы таблицами значений.*

Упражнение 14 Пусть c_n есть максимум сложности $c(f)$ по всем булевым функциям f от n переменных. Докажите, что $1,99^n < c_n < 2,01^n$ при достаточно больших n .

Упражнение 15 Покажите, что любую функцию можно вычислить схемой глубины не более 3 из элементов NOT и из элементов AND и OR с произвольным числом входов.

Упражнение 16 Докажите, что если из схемы глубины $O(\log n)$, вычисляющей функцию $f: \{0,1\}^n \rightarrow \{0,1\}^m$, выбросить все несущественные присваивания, то полученная схема имеет полиномиальный по $n + m$ размер.

Упражнение 17 Постройте схему, которая сравнивает два n -битовых числа и имеет размер $O(n)$, а глубину $O(\log n)$.

Упражнение 18 1. Постройте схему сложения двух n -битовых чисел размера $O(n)$.

2. Тот же вопрос, если дополнительно потребовать, чтобы глубина схемы была $O(\log n)$.

Упражнение 19 Функция $MAJ \{0,1\}^n \rightarrow \{0,1\}$ равна 1 на двоичных словах, в которых число единиц больше числа нулей, и 0 — на остальных словах. Постройте схему, вычисляющую эту функцию, размер схемы должен быть линеен по n , глубина — $O(\log n \log \log n)$.

Упражнение 20 Постройте схему размера $\text{poly}(n)$ и глубины $O(\log^2 n)$, которая проверяет, связаны ли путём две вершины в графе. Граф на m вершинах, которые помечены числами от 1 до m , задаётся $n = m(m-1)/2$ булевыми переменными. Переменная x_{ij} , где $i < j$, определяет, есть ли в графе ребро, соединяющее вершины i и j .

Упражнение 21 Пусть схема глубины 3 из элементов NOT и из элементов AND и OR с произвольным числом входов вычисляет сложение n битов по модулю 2 (функция $PARITY$). Покажите, что размер схемы не меньше c^n для некоторого $c > 1$.

Упражнение 22 Пусть f_1, f_2, \dots — последовательность булевых функций от $1, 2, \dots$ аргументов. Покажите, что следующие два свойства равносильны:

1. существует последовательность вычисляющих эти функции формул, размер которых не превосходит полинома от n ;
2. существует последовательность вычисляющих эти функции схем глубины $O(\log n)$ из элементов NOT , AND и OR (с двумя входами).

Упражнение 23 Докажите, что существует разрешимый предикат, который принадлежит P/poly , но не принадлежит P .

1.6 Полиномиальные сводимости и \mathbf{NP} -полнота

Алгоритмическая задача называется *труднорешаемой*, если для нее не существует полиномиального алгоритма.

Известно, что бывают алгоритмические задачи *в принципе* неразрешимые вообще никаким алгоритмом (См. например задачу 5). Поэтому естественно задаться вопросом, а существуют ли *разрешимые* задачи, которые тем не менее не принадлежат классу \mathbf{P} . Ответ на этот вопрос предоставляется *теоремой об иерархии* [Дж.67], которая наряду с теоремой Блума об ускорении является одним из краеугольных камней теории сложности вычислений. Так же как и в случае с теоремой об ускорении, мы приводим ее упрощенный вариант:

Теорема 9 *Существует алгоритмическая задача, разрешимая некоторым алгоритмом сложности $n^{O(\log n)}$, но не принадлежащая классу \mathbf{P} .*

К сожалению, задачи возникающие как в теореме об ускорении, так и в теореме об иерархии носят довольно искусственный характер и по этой причине не могут быть использованы для сложностного анализа переборных задач дискретной оптимизации. Для этой цели используется теория \mathbf{NP} -полноты, изложение основ которой мы начинаем с понятия *полиномиальной сводимости*.

1.6.1 Сводимость по Куку

На неформальном уровне мы уже познакомились с этим понятием в разделе 1.1:

Определение 17 *Алгоритмическая задача P_1 полиномиально сводится к задаче P_2 , если существует полиномиальный алгоритм для решения задачи P_1 , возможно, вызывающий в ходе своей работы процедуру для решения задачи P_2 .*

При таком определении, однако, возникает один неприятный эффект. Например, алгоритм 7 вне всякого сомнения полиномиален, если последний оператор $R \leftarrow R \cdot R$ понимать как вызов процедуры умножения. Далее, мы уже отмечали в разделе 1.2.1, что для умножения также существует полиномиальный алгоритм. Тем не менее составной алгоритм 7 неполиномиален и, более того, вычисляемая им функция 2^{2^t} по очевидным причинам не принадлежит классу \mathbf{P} . Таким образом, класс \mathbf{P} оказывается незамкнутым относительно полиномиальной сводимости что, разумеется, ненормально. Понятны и причины этого: при повторных обращениях к функциональным процедурам может происходить лавинообразный рост значений числовых параметров. Наиболее кардинальный и в то же время общепринятый способ борьбы с этим явлением — с самого начала ограничиться рассмотрением лишь задач разрешения, т.е. таких, ответ на которые имеет вид ДА/НЕТ. При таком ограничении в полиномиальных сводимостях используются только предикатные процедуры, указанная выше проблема снимается и класс \mathbf{P} уже будет замкнутым относительно полиномиальной сводимости.

Прежде чем двигаться дальше, отметим, что любую переборную задачу дискретной оптимизации можно без ограничения общности заменить на переборную же задачу разрешения, так что рассматриваемое ограничение не слишком обременительно. Мы проиллюстрируем как осуществляется эта замена на примере задачи КОММИВОЯЖЕР, хотя видно, что тот же самый метод годится для любой задачи дискретной оптимизации.

Именно, давайте вместо *нахождения* минимального значения суммы (1.1) ограничимся более простым *вопросом* "верно ли, что $\text{Comm}(m, d) \leq B$ ", где $\text{Comm}(m, d)$ — минимальное возможное значение суммы (1.1), где B — новый числовой параметр. Более формально,

Задача 6 КОММИВОЯЖЕР-разрешимость. Заданы n городов c_1, c_2, \dots, c_n и попарные расстояния $d_{ij} \equiv d(c_i, c_j)$ между ними, являющиеся положительными целыми числами, и положительное целое B .

Верно ли, что минимально возможное значение суммы (1.1) меньше B ?

Тогда задача КОММИВОЯЖЕР допускает следующее простое полиномиальное сведение к этой задаче разрешения, осуществляемое с помощью *бинарного поиска* (алгоритм 9).

Алгоритм 9 Сведение оптимизационных задач к задачам разрешения

Вход: $m > 0$, $d_{ij} > 0$, $\forall i, j \in (1, \dots, m)$

Выход: Перестановка π_{opt} , такая, что (1.1) минимальна.

{ Используется процедура $TSP_{bool}(m, D, B)$ для задачи 6 }

$B_{\min} \leftarrow 0$

$B_{\max} \leftarrow m \cdot \max_{1 \leq i < j \leq m} d(c_i, c_j)$ { $B_{\min} \leq (1.1) \leq B_{\max}$, где $TSP_{bool}(m, d)$ }

while $B_{\max} - B_{\min} > 1$ **do**

$B \leftarrow \lfloor (B_{\min} + B_{\max})/2 \rfloor$

if $TSP_{bool}(m, D, B)$ **then**

$B_{\max} \leftarrow B$

else

$B_{\min} \leftarrow B$

end if

end while

if $TSP_{bool}(m, D, B_{\min})$ **then**

return B_{\min}

else

return B_{\max}

end if

В теории сложности вычислений, „процедурный“ вариант полиномиальной сводимости называется *Тьюринговой сводимостью* или *сводимостью по Куку*, в честь автора работы [Кук75b].

1.6.2 Недетерминированные алгоритмы

Математическим уточнением понятия "переборная задача разрешения" служит "задача, разрешимая за полиномиальное время с помощью недетерминированного полиномиального алгоритма".

В *недетерминированных алгоритмах* дополнительно разрешаются *недетерминированные операторы перехода* вида

goto ℓ_1 **or** ℓ_2 .

Таким образом, для каждого массива входных данных имеется не один, а несколько (в общем случае — экспоненциальное число) путей, по которым может развиваться вычисление. Недетерминированный алгоритм по определению выдает окончательный ответ 1, если *существует хотя бы один* путь развития вычисления, на котором выдается ответ 1, и 0 — в противном случае (таким образом, ответы ДА и НЕТ в случае недетерминированных вычислений несимметричны).

Более формально,

Определение 18 *Недетерминированная Машина Тьюринга (НМТ) это набор $T = \langle k, \Sigma, \Gamma, \Phi \rangle$, где $k \geq 1$ — натуральное число, Σ, Γ — конечные множества, $\star \in \Sigma$, $START, STOP \in \Gamma$, а Φ — произвольное отношение*

$$\Phi \subset (\Gamma \times \Sigma^k) \times (\Gamma \times \Sigma^k \times \{-1, 0, 1\}^k).$$

Переход из состояния g , с символами на лентах h_1, \dots, h_k будет допустим, если новое состояние g' , записанные символы h'_1, \dots, h'_k и смещения головок $\varepsilon_1, \dots, \varepsilon_k$ удовлетворяют соотношению :

$$(g, h_1, \dots, h_k, g', h'_1, \dots, h'_k, \varepsilon_1, \dots, \varepsilon_k) \in \Phi.$$

Процесс вычисления недетерминированной машиной Тьюринга можно представлять таким образом, что в каждый момент "ветвления—совершения недетерминированного перехода из нескольких возможных, происходит "клонирование" НМТ, и вычисление продолжается по всем возможным путям, пока одной из "клонированных" НМТ не будет получен утвердительный ответ (1).

Очевидно, детерминированные машины Тьюринга являются частным случаем недетерминированных.

Напомним, что язык L называется *рекурсивно перечислимый*, если $L = \emptyset$ или существует вычислимая функция f такая, что область значений f совпадает с L .

Сложностью (временем работы) недетерминированного алгоритма (НМТ) на входном слове x называется *минимальная* сложность вычисления, приводящего на этом входе к ответу 1.

Зная, что такое время работы НМТ для каждого входного слова x , можно аналогично определению 4 ввести для НМТ и *сложность в наихудшем случае*¹⁹. Аналогично определению классов $DTIME(f(n))$, $DSPACE(f(n))$ (См. определения 6, 9) определим классы $NTIME(f(n))$, $NSPACE(f(n))$.

Важно отметить серьезное отличие классов сложности языков для НМТ и детерминированных МТ. Это замкнутость классов сложности языков относительно дополнения для детерминированных МТ, и отсутствие этого свойства для НМТ.

Т.е., для любого класса $DTIME(f(n))$, класс

$$co-DTIME(f(n)) \equiv \{L | \bar{L} \in DTIME(f(n))\},$$

состоящий из дополнений языков²⁰ совпадает с классом $DTIME(f(n))$. Действительно, для любого языка $L \in DTIME(f(n))$ существует детерминированная машина Тьюринга T с временной сложностью $f(n)$, то из нее легко (инвертировав окончательный ответ) получить машину \bar{T} , также принадлежащую классу $DTIME(f(n))$. Аналогично показывается замкнутость относительно дополнений и классов пространственной сложности детерминированных машин Тьюринга — $DSPACE(f(n))$.

Для классов $NTIME(f(n))$ и $NSPACE(f(n))$ эти рассуждения не проходят из-за асимметрии положительных и отрицательных ответов в определении сложности НМТ, и таким образом нельзя утверждать о совпадении классов $NTIME(f(n))$ и $co-NTIME(f(n))$ ($NSPACE(f(n))$ и $co-NSPACE(f(n))$), хотя обратное и не доказано.

Теперь, определим важный в теории сложности класс задач разрешения (языков), разрешимых *полиномиальными недетерминированными алгоритмами* и его ко-класс:

Определение 19

$$\begin{aligned} NP &= \cup_{c>0} NTIME(n^c), \\ co-NP &= \{L | \bar{L} \in NP\}. \end{aligned}$$

Упражнение 24 Покажите, что $P \subseteq NP \cap co-NP$.

Следует подчеркнуть, что недетерминированные алгоритмы представляют собой лишь некоторую довольно удобную математическую абстракцию и в отличие от детерминированных и вероятностных алгоритмов не соответствуют реально существующим вычислительным или аналоговым машинам (хотя крайне интересные и перспективные исследования в области генетических алгоритмов и квантовых автоматов, позволяют в очень осторожной форме высказать предположение о том, что в обозримом будущем ситуация может измениться).

¹⁹ При желании можно определить *сложность в среднем*, но такая мера и используется крайне редко

²⁰ Определение дополнения языка можно найти в разделе 5

В частности, недетерминированные полиномиальные алгоритмы вообще говоря не являются эффективными, хотя и известно включение $NP \subseteq PSPACE$.

Далее, все переборные задачи лежат в классе NP . Например, КОММИВОЯЖЕР в форме задачи разрешения (задача 6) решается нижеследующим недетерминированным полиномиальным алгоритмом 10.

Алгоритм 10 Моделирование перебора недетерминизмом

Вход: $m > 0$, $d_{ij} > 0$, $\forall i, j \in (1, \dots, m), B$

Выход: "Да", если существует перестановка π_{opt} , такая, что (1.1) $\leq B$.

```

for all  $i \in 1..m$  do
   $\pi_i \leftarrow 0$ 
  for all  $j \in 0..\lceil \log_2 m \rceil$  do
    goto  $\ell_0$  or  $\ell_1$ 
    label  $\ell_1$ :
       $\pi_i \leftarrow \pi_i + 2^j$ 
    label  $\ell_0$ :
  end for
  if  $\pi_i > m$  then
    return НЕТ {эта недетерминированная часть "угадывает"  $\pi_i \in \{1, \dots, m\}$ }
  end if
end for
if  $\pi_1, \dots, \pi_m$  попарно различны and  $\sum_{i=1}^m d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B$  then
  then return ДА
else
  return НЕТ
end if

```

Справедливо и обратное: вычисление с помощью любого недетерминированного полиномиального алгоритма можно представить как переборную задачу (перебор ведется по всем двоичным словам, кодирующим направления ветвлений в недетерминированных операторах перехода). Тем самым, "задача из класса NP " оказывается адекватным математическим уточнением понятия "переборная задача разрешения".

Более формально это формулируется в одном из альтернативных определений класса NP , через детерминированные машины Тьюринга.

Определение 20 Язык $L \subseteq \Sigma^*$ принадлежит классу NP , если существует полиномиальный алгоритм A и некоторый полином q такие, что $L = \{x \in \Sigma^* : \exists y, |y| < q(|x|) \& A(x, y) = 1\}$.

Слово y называется обычно "подсказкой".

Например, пусть L описывает задачу КОММИВОЯЖЕР в форме задачи разрешения 6. Тогда слово x представляет собой матрицу расстояний между городами и B — денежное ограничение затрат на посещение всех городов, а слово y будет

представлять объезд всех городов с затратами $< B$. Проверить, что путь y при параметрах задачи x , действительно является решением, можно полиномиальным алгоритмом. С другой стороны, если для задачи с некоторыми параметрами x' , такого пути ($< B$) нет, то, соответственно, такой подсказки $y(x')$ не существует.

Через определение 20 еще легче увидеть "переборную" сущность класса NP , т.к. видно, что можно перебирать детерминированным алгоритмом множество вариантов, эффективно (полиномиальным алгоритмом) проверяя каждый вариант.

Теорема 10 Определения 19 и 20 эквивалентны.

Доказательство (19) \rightarrow (20). Для каждого слова $x \in L$, подсказкой y можно назначить закодированный кратчайший протокол выполнения-подтверждения НМТ T , определяющей язык L , в смысле определения 19. Так как длина кратчайшего пути-подтверждения $T(x)$ полиномиально ограничена, то можно выбрать полиномиально ограниченную кодировку $y(x)$. Проверить целостность протокола (отсутствие противоречий с определением НМТ T) $y(x)$ можно также за полиномиальное время.

(20) \rightarrow (19). Еще проще. Пусть НМТ T , недетерминированно дописывает дописывает разделитель $\#$ и некоторое слово y , к входному слову x , а затем работает над словом $x\#y$, как полиномиальная детерминированная машины Тьюринга из определения 20. \square

1.6.3 Сводимость по Карпу

Понятие полиномиальной сводимости по Куку (определение 17) оказывается чрезвычайно общим для изучения переборных задач ввиду отмеченной выше асимметрии ответов 1 и 0.

Например, задача разрешения " $\text{Comm}(m, d) > B$ " (в которой спрашивается верно ли, что *любой* маршрут коммивояжера имеет длину по крайней мере $(B + 1)$), принадлежит классу $co-NP$, и не принадлежит классу NP , при общепринятой гипотезе $P \neq NP$. В то же время она очевидным образом сводится по Куку к переборной задаче 6, принадлежащей классу NP .

Поэтому в теории сложности вычислений гораздо большее распространение получил ограниченный вариант полиномиальной сводимости по Куку, впервые рассмотренный в основополагающей работе [Кар75a] и соответственно называемый иногда *сводимостью по Карпу* (мы будем использовать для нее просто термин *полиномиальная сводимость*, также широко распространенный в литературе).

Определение 21 Задача разрешения P_1 полиномиально сводится к задаче разрешения P_2 , если существует полиномиально вычислимая функция f , перерабатывающая массивы входных данных I_1 для задачи P_1 в массивы входных данных $I_2 \equiv f(I_1)$ для задачи P_2 таким образом, что для любого I ответ в задаче P_1 совпадает с ответом задачи P_2 для входных данных $f(I)$.

Легко видеть, что относительно такого усеченного варианта полиномиальной сводимости, класс переборных задач разрешения NP уже оказывается замкнутым.

Фундаментальной открытой проблемой теории сложности вычислений (а к настоящему времени и одной из наиболее фундаментальных проблем всей современной математики — [Sma00]) является вопрос о совпадении классов сложности P и NP . Иными словами, все ли переборные задачи можно решить с помощью эффективного алгоритма? (в связи с этой емкой формулировкой $P \stackrel{?}{=} NP$ -проблему еще называют иногда *проблемой перебора*.)

На первый взгляд $P \stackrel{?}{=} NP$ -проблема по существу представляет собой совокупность похожих, но формально между собой не связанных вопросов о том, поддается ли эффективному решению *данная конкретная* переборная задача. Тем не менее оказывается, что рассмотренное выше понятие полиномиальной сводимости позволяет во многих случаях установить, что эти вопросы для целого ряда задач эквивалентны между собой, а также эквивалентны "глобальной" задаче $P \stackrel{?}{=} NP$.

Более точно,

Определение 22 *Задача разрешения называется NP -полной ²¹, если она сама принадлежит классу NP , а с другой стороны произвольная задача из этого класса сводится к ней полиномиально (по Карпу).*

Оставляя на секунду в стороне неочевидный а priori вопрос о существовании хотя бы одной NP -полной задачи, заметим, что они в точности обладают нужным нам свойством: $P = NP$ тогда и только тогда, когда *некоторую* NP -полную задачу можно решить с помощью полиномиального алгоритма. Иными словами, NP -полные задачи выполняют роль наиболее сложных, универсальных задач в классе NP , и все они по своей сложности эквивалентны между собой.

Историю современной теории сложности вычислений принято отсчитывать с работ [Кук75b, Кар75a], в которых были заложены основы теории NP -полноты и доказано существование вначале одной, а затем (в работе [Кар75a]) достаточно большого числа (а именно, 21) естественных NP -полных задач. В неоднократно цитировавшейся монографии [М. 82] (вышедшей на английском языке в 1979 г.) приводится список известных к тому времени NP -полных задач, насчитывающий уже более 300 наименований. К настоящему времени количество известных NP -полных задач выражается четырехзначным числом, и постоянно появляются новые, возникающие как в самой математике и теории сложности, так и в таких дисциплинах как биология, социология, военное дело, теория расписаний, теория игр и т.д. Более того, как мы уже отмечали в разделе 1.1 на интуитивном уровне, для подавляющего большинства задач из класса NP в конечном итоге удастся либо установить их принадлежность классу P (т.е. найти полиномиальный алгоритм) либо доказать NP -полноту. Одним из наиболее важных исключений являются задачи типа дискретного логарифма, на которых основаны современные криптосистемы.

²¹ Чтобы не перегружать лекции излишней терминологией, мы будем называть в дальнейшем оптимизационную задачу NP -полной, если NP -полна соответствующая задача разрешения.

Именно этим обстоятельством объясняется важность проблемы перебора $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. Безуспешным попыткам построения полиномиальных алгоритмов для **NP**-полных задач были посвящены усилия огромного числа выдающихся специалистов в данной области (в конце раздела мы вкратце расскажем о некоторых частичных результатах, полученных в обратном направлении, т.е. попытках доказать неравенство $\mathbf{P} \neq \mathbf{NP}$). Ввиду этого можно считать, что **NP**-полные задачи являются *труднорешаемыми* со всех практических точек зрения, хотя, повторяем, строгое доказательство этого составляет одну из центральных открытых проблем современной математики.

Изучением более глубоких взаимоотношений между вероятностными, недетерминированными и др. типами вычислений, различными видами сводимостей и т.д. занимается раздел теории сложности, называемый *структурной сложностью*. Наше же краткое знакомство с этим предметом заканчивается; для практически ориентированного читателя (по причинам, объясненным в разделе 1.1) гораздо важнее иметь в своих руках удобные и надежные инструменты, с помощью которых он сможет сам устанавливать **NP**-полноту интересующих его задач.

Прежде всего необходимо иметь хотя бы одну такую задачу. Честь быть первой выпала в работе [Кук75b] на роль следующей задачи **ВЫПОЛНИМОСТЬ**:

Задача 7 **ВЫПОЛНИМОСТЬ**²²

Дано булевское выражение, являющееся **конъюнктивной нормальной формой** (КНФ)

$$\bigwedge_{i=1}^m K_i, \quad (1.6)$$

где K_i — элементарные дизъюнкции вида

$$x_{j_1}^{\sigma_1} \vee \dots \vee x_{j_k}^{\sigma_k}, \quad (1.7)$$

$\sigma_j \in \{0, 1\}$, $x^1 = x$ и $x^0 = (\neg x)$.

Существует ли (булевский) набор переменных x_j , обращающий эту форму в 1 (т.е. **TRUE**)?

Теорема 11 Задача 7 (**ВЫПОЛНИМОСТЬ**) — **NP**-полна.

Доказательство Очевидно, **ВЫПОЛНИМОСТЬ** принадлежит **NP**, так как для любого слова x , представляющего выполнимую входную КНФ, существует подсказка y — значения переменных, при которых эта КНФ выполняется, причем проверку легко выполнить за полиномиальное время.

Рассмотрим произвольный язык $L \in \mathbf{NP}$. Согласно определению 20,

$$\forall x \in L, \exists y(x) : |y(x)| < \text{poly}(|x|),$$

²²В англоязычной литературе — Satisfiability или просто SAT

и существует МТ \mathcal{M} , распознающая $L_y = \{x\#y(x) | x \in L\}$ за полиномиальное время.

Рассмотрим таблицу вычисления (см. доказательство теоремы 7) для \mathcal{M} .

Будем использовать те же переменные, что и в доказательстве теоремы 7 (коды состояний клеток таблицы вычисления). Чтобы таблица вычисления соответствовала правильно проведённому успешному (с ответом 1) вычислению, должны выполняться локальные правила согласования для каждой четвёрки клеток вида $\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}$ и результат должен быть 1. Каждое такое правило задаётся формулой от переменных, отвечающих либо рассматриваемой четвёрке, либо нулевой ячейке самой нижней строки таблицы. Определим формулу φ_x как конъюнкцию всех этих формул, в которые подставлены значения переменных, кодирующих вход $x\#y$, дополненный символами \star до длины $|x| + 1 + q(|x|)$. Значения, соответствующие x и $\#$, — константы, поэтому переменные, от которых зависит эта формула, отвечают y и кодам внутренних ячеек таблицы. Так что можно считать, что формула φ_x зависит от y и ещё от каких-то переменных, которые мы обозначим z .

Итак, мы сопоставили слову x формулу $\varphi_x(y, z)$, которая по построению обладает следующим свойством. Если \mathcal{M} распознает $x\#y$, то найдётся такой набор значений $z(x, y)$, при котором $\varphi_x(y, z(x, y))$ истинна (эти значения описывают работу \mathcal{M} на входе $x\#y$). А если \mathcal{M} не распознает $x\#y$, то $\varphi_x(y, z)$ всегда ложна (поскольку по сути утверждает, что вычисление на входе (x, y) даёт ответ 1). Таким образом, при $x \in L$ такая формула иногда (при некоторых значениях y) истинна, при $x \notin L$ — всегда ложна.

□

Все остальные доказательства **NP**-полноты (а уже следующая работа [Кар75a] содержала 21 такое доказательство) основаны на следующем легко проверяемом замечании: если **NP**-полная задача P_1 полиномиально сводится к переборной задаче P_2 , то P_2 также **NP**-полна. Поэтому общие рекомендации состоят в том, чтобы выбрать в списке уже известных **NP**-полных задач "максимально похожую" и попытаться свести ее к интересующей нас задаче.

К сожалению, о том как именно применять эту общую рекомендацию в каждом конкретном случае что-либо определенное сказать довольно трудно. Доказательства **NP**-полноты являются скорее искусством, и мы от всей души желаем нашим читателям больше полиномиальных алгоритмов, хороших и разных, для интересующих их задач с тем, чтобы обращаться к этому специфическому искусству им пришлось как можно реже. В случае, если такая необходимость все же возникнет, мы рекомендуем обратиться к прекрасно написанной главе 3 монографии [М. 82], по которой можно ознакомиться по крайней мере с некоторыми ориентирами на этом пути. Здесь же мы ограничимся простым примером.

Задача 8 3-ВЫПОЛНИМОСТЬ²³ *Вариант задачи ВЫПОЛНИМОСТЬ, в котором каждая элементарная дизъюнкция (1.7) имеет длину $k \leq 3$ (соответствующие КНФ называются 3-КНФ).*

²³В англоязычной литературе — 3SAT

Оказывается, **ВЫПОЛНИМОСТЬ** сводится к своему ограниченному варианту **3-ВЫПОЛНИМОСТЬ**. В самом деле, если дана некоторая КНФ (1.6), мы заменяем в ней каждую элементарную дизъюнкцию (1.7) с $k > 3$ на следующее булевское выражение:

$$(y_{i2} \equiv (x_{j_1}^{\sigma_1} \vee x_{j_2}^{\sigma_2})) \wedge (y_{i3} \equiv (y_{i2} \vee x_{j_3}^{\sigma_3})) \wedge \dots \wedge (y_{ik} \equiv (y_{i,k-1} \vee x_{j_k}^{\sigma_k})) \wedge y_{ik},$$

где y_{i2}, \dots, y_{ik} — новые булевы переменные, и трансформируем эквивалентности $y_{i\nu} \equiv (y_{i,\nu-1} \vee x_{j_\nu}^{\sigma_\nu})$ в 3-КНФ стандартным способом. Во всяком выполняющем наборе для полученной таким образом 3-КНФ переменная $y_{i\nu}$ обязательно должна принять значение $(x_{j_1}^{\sigma_1} \vee \dots \vee x_{j_\nu}^{\sigma_\nu})$ и, в частности, y_{ik} получает значение (1.7). Поэтому наше преобразование определяет полиномиальную сводимость задачи **ВЫПОЛНИМОСТЬ** к задаче **3-ВЫПОЛНИМОСТЬ**, и последняя тем самым оказывается **NP**-полной.

Упражнение 25 Преобразуйте отношение **ЭКВИВАЛЕНТНОСТЬ** в 3-КНФ.

Теперь рассмотрим максимально ограниченную версию фундаментальной задачи о покрытии — **ВЕРШИННОЕ ПОКРЫТИЕ**, и покажем что даже она **NP**-полна.

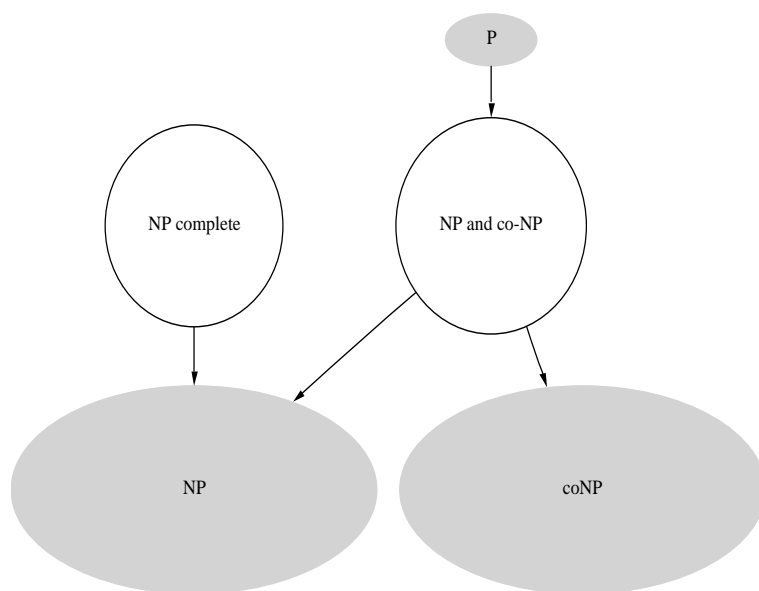
Задача 9 **ВЕРШИННОЕ ПОКРЫТИЕ**²⁴

Дан граф $G = (V, E)$ (т.е. некоторое множество вершин V , некоторые пары которых соединены дугами из E) и положительное целое число K , $K \leq |V|$.

Имеется ли в графе G **вершинное покрытие** не более, чем из K элементов, т.е. такое подмножество $V' \subseteq V$, что $|V'| \leq K$ и каждая дуга из E содержит хотя бы одну вершину из V' ?

Мы покажем, что **3-ВЫПОЛНИМОСТЬ** полиномиально сводится к **ВЕРШИННОМУ ПОКРЫТИЮ**. Пусть имеется 3-КНФ (1.6) от n переменных x_1, \dots, x_n , в которой (без существенного ограничения общности) можно считать, что $k = 3$ для всех элементарных дизъюнкций (1.7). Для каждой переменной x_j мы вводим отдельную дугу $(x_j, \neg x_j)$, а для каждой элементарной дизъюнкции (1.7) — треугольник с вершинами (v_{i1}, v_{i2}, v_{i3}) , после чего соединяем дугами v_{i1} с $x_{j_1}^{\sigma_1}$, v_{i2} с $x_{j_2}^{\sigma_2}$ и v_{i3} с $x_{j_3}^{\sigma_3}$. Тогда в полученном графе всякое вершинное покрытие должно иметь размер не менее $(n + 2m)$ (по крайней мере n вершин нужно, чтобы покрыть ребра $(x_j, \neg x_j)$ и по крайней мере $2m$ необходимо для покрытия треугольников (v_{i1}, v_{i2}, v_{i3})), причем легко видеть, что покрытия такого размера в точности соответствуют выполняющим наборам исходной 3-КНФ. Таким образом, если мы положим параметр K в формулировке задачи **ВЕРШИННОЕ ПОКРЫТИЕ** равным $(n + 2m)$, мы получим искомое полиномиальное сведение. Следовательно, задача **ВЕРШИННОЕ ПОКРЫТИЕ** **NP**-полна.

²⁴В англоязычной литературе — Vertex Covering

Рис. 1.6: Иерархия классов P , NP , $coNP$.

В настоящее время "экспертное мнение" склоняется к тому, что скорее всего все же $P \neq NP$. Таким образом можно представить общепринятое представление об иерархии классов следующим образом (Рис 1.6).

Для того чтобы доказать этот факт, необходимо научиться получать *нижние оценки* сложности *любого* алгоритма, предназначенного для решения некоторой задачи из класса NP . Соответствующая проблема так и называется *проблемой нижних оценок*. Мы уже видели ранее (и познакомимся с массой таких примеров в дальнейшем), что для самых разнообразных задач возможны весьма неожиданные алгоритмы, основанные на самых различных идеях. Для доказательства же нижних оценок необходимо найти некоторое *общее* рассуждение, которое учло бы все такие алгоритмы, как уже существующие, так и те, которые могут быть построены в будущем. Поэтому проблема нижних оценок является крайне трудной, и решена она лишь в довольно частных случаях.

Традиционно эта проблема в подавляющем большинстве случаев рассматривается в контексте *схемной* (или *булевой*) сложности (См. раздел 1.5), ввиду исключительной внешней простоты и наглядности схемной модели. Из отмеченной взаимосвязи между алгоритмами и схемами вытекает, что проблема нижних оценок для последних должна быть столь же трудной как и для алгоритмов. Действительно, наилучшая известная оценка сложности схем для какой-либо задачи из класса NP всего лишь линейна от числа переменных n и составляет $(5n - 1)$ [Ред71].

Гораздо больших успехов в получении нижних оценок сложности удалось добиться для схем с различными ограничениями. К числу таких ограниченных моделей относятся, например, *монотонные схемы* (в которых запрещаются элементы *NOT*) или *схемы ограниченной глубины* (в которых вдоль любого пути число че-

редований элементов различных типов ограничено сверху произвольно большой, но фиксированной заранее константой). Для таких схем проблема нижних оценок практически полностью решена: см., например, [Раз85b, Раз85а, Раз85с, RW90].

С другой стороны, работа над верхними оценками — построение конкретных алгоритмов, как правило, производится в терминах машин со случайным доступом или машин Тьюринга.

Упражнение 26 *Покажите, что задача распознавания гамильтоновых графов (т.е. графов, содержащих гамильтонов цикл), принадлежит NР.*

Упражнение 27 *Покажите, что задача распознавания негамильтоновых графов (т.е. графов, не содержащих ни одного гамильтонова цикла), принадлежит Co-NР.*

Глава 2

Приближенные алгоритмы с гарантированными оценками точности

2.1 Жадный алгоритм в задаче о покрытии

Задача 10 *Покрывание множества*¹

Пусть на m -элементном множестве X задано некоторое семейство его подмножеств $F = \{S_1, \dots, S_n\}$ и

$$X = \cup_{i=1}^n S_i$$

Надо найти минимальное по числу подмножеств подсемейства $P \subseteq F$, обладающее свойством покрытия, то есть нахождения минимального $J \subseteq \{1, 2, \dots, n\}$ такого, что

$$X = \cup_{i \in J} S_i \quad (2.1)$$

Число $|J|$ называется размером минимального покрытия.

Известно, что задача о покрытии NP-полна. По этой причине трудно надеяться на существование полиномиального алгоритма ее решения. Одним из общих подходов к решению NP-трудных задач бурно развивающимся в настоящее время является разработка приближенных алгоритмов с гарантированными оценками качества получаемого решения.

Определение 23 *Приближенный алгоритм является алгоритмом с мультипликативной точностью D , если он при любых исходных данных находит допустимое решение со значением целевой функции, отличающемся от оптимума не более, чем в D раз.*

¹Set covering

Одной из простейших эвристик часто применяемых при решении различных задач является так называемая жадная эвристика. Применительно к задаче о покрытии она заключается в выборе на каждом шаге подмножества, покрывающего максимальное число еще непокрытых элементов. Эффективность описанного приближенного алгоритма для задачи о покрытии очевидна.

Оценим сейчас точность, которую жадный алгоритм гарантирует в задаче о покрытии.

Пусть X_k — число непокрытых после k -го шага, M — размер минимального покрытия.

Имеем:

$$X_{k+1} \leq X_k - \frac{X_k}{M} = X_k(1 - 1/M). \quad (2.2)$$

Упражнение 28 Докажите неравенство (2.2).

Отсюда, учитывая, что $|X_0| = m$ получаем:

$$X_k \leq m(1 - 1/M)^k \leq m \exp(-\frac{k}{M})$$

Найдем наибольшее k_0 при котором

$$m \exp(-\frac{k_0}{M}) \geq 1.$$

Тогда при $k_1 = k_0 + 1$

$$X_{k_1} < 1,$$

что означает, что все элементы покрыты.

При этом

$$k_1/M \leq 1 + \ln m,$$

значит размер покрытия построенного жадным алгоритмом превосходит минимальное не более, чем в $1 + \ln m$ раз.

Упражнение 29 Постройте пример, где оценка $O(1 + \ln m)$ размера покрытия построенного жадным алгоритмом достигается по порядку. Указание: достаточно рассмотреть случай, когда размер минимального покрытия $M = 2$.

Упражнение 30 Постройте пример, где эта оценка достигается асимптотически.

2.2 Алгоритм Кристофидеса для метрической задачи коммивояжера

Напомним некоторые определения.

Определение 24 *Эйлеровым путем в графе называется произвольный путь, проходящий через каждое ребро графа в точности один раз.*

Определение 25 *Замкнутый эйлеров путь называется эйлеровым обходом или эйлеровым циклом.*

Определение 26 *Эйлеров граф — граф, в котором существует эйлеров обход.*

Приведем критерий эйлеровости графа.

Теорема 12 *Эйлеров обход в графе существует тогда и только тогда, когда граф связный и все его вершины четной степени.*

Доказательство Доказательство достаточности условия теоремы будет следствием анализа алгоритма нахождения эйлерова пути (алгоритм 11).

Необходимость условия очевидна, так как если некоторая вершина v появляется в эйлеровом обходе k раз, то это означает, что степень этой вершины в графе составляет $2k$. \square

Алгоритм 11 Алгоритм нахождения эйлерова обхода

Вход: Граф $G(V, E)$, связный, с вершинами четной степени.

Выход: Эйлеров обход для графа G .

$\{v_0$ — произвольная вершина в $G\}$

$G^*(V^*, E^*) \leftarrow G$.

return $(Euler(v_0))$.

PROCEDURE $Euler(vertice\ v_0)$

if все соседние вершины v_0 в G отсутствуют в G^* **then**

return пустой цикл;

end if

{Если $v_0 \notin V^*$, то $V^* \leftarrow V^* \cup v_0$ }

{Начиная с вершины v_0 строим замкнутый обход (v_0, \dots, v_k, v_0) графа G^* , выбирая произвольно непосещенные вершины в V^* .}

{Удаляем вершины (v_0, \dots, v_k) из G^* }

return $Euler(v_0) || Euler(v_1) || \dots || Euler(v_k)$. {|| — конкатенация циклов.}

END PROCEDURE

Упражнение 31 Покажите, что сложность алгоритма 11 (нахождение эйлера обхода) есть $O(|E|)$.

Определение 27 Задача КОММИВОЯЖЕР (Зад. 2) называется метрической, если для матрицы расстояний выполнено неравенство треугольника:

$$\forall i, j, k \quad d_{ik} \leq d_{ij} + d_{jk}$$

Заметим, что здесь рассматривается полный граф.

Необходимо отметить, что метрическая задача коммивояжера NP-полна. Это легко доказать, заметив что если веса ребер полного графа принимают только два значения 1 и 2, то задача является метрической. В свою очередь, построение минимального обхода здесь эквивалентно ответу на вопрос: существует ли в графе с вершинами исходного графа и ребрами, имеющими вес 1, гамильтонов цикл?

Для этой вариации задачи можно предложить следующий приближенный алгоритм 12 с мультипликативной точностью 2.

Пример. В рассмотренном на рисунке примере эйлеров цикл составляет следующий маршрут:

$$\begin{aligned} &v_0 \rightarrow v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_8 \rightarrow v_9 \rightarrow v_8 \rightarrow v_9 \rightarrow v_{13} \rightarrow v_{12} \rightarrow v_{13} \\ &\rightarrow v_9 \rightarrow v_{10} \rightarrow v_{14} \rightarrow v_{15} \rightarrow v_{14} \rightarrow v_{10} \rightarrow v_{11} \rightarrow v_{10} \rightarrow v_6 \rightarrow v_7 \rightarrow v_6 \rightarrow v_2 \rightarrow v_3 \\ &\rightarrow v_2 \rightarrow v_6 \rightarrow v_{10} \rightarrow v_9 \rightarrow v_5 \rightarrow v_1 \rightarrow v_0 \end{aligned}$$

Соответствующий ему вложенный тур (гамильтонов цикл) таков:

$$[v_0, v_1, v_5, v_4, v_9, v_8, v_{13}, v_{12}, v_{10}, v_{14}, v_{15}, v_{11}, v_6, v_7, v_2, v_3, v_0].$$

Теорема 13 Отношение длины пути, построенного алгоритмом 12 к длине оптимального пути, не превосходит 2.

Доказательство Длина кратчайшего гамильтонова пути не меньше длины кратчайшего остовного дерева. Длина эйлерова маршрута в G равна удвоенной длине минимального остовного дерева (по построению). По неравенству треугольника, полученный гамильтонов цикл имеет длину не превосходящую длину эйлерова обхода, т.е. удвоенную длину минимального остовного дерева.

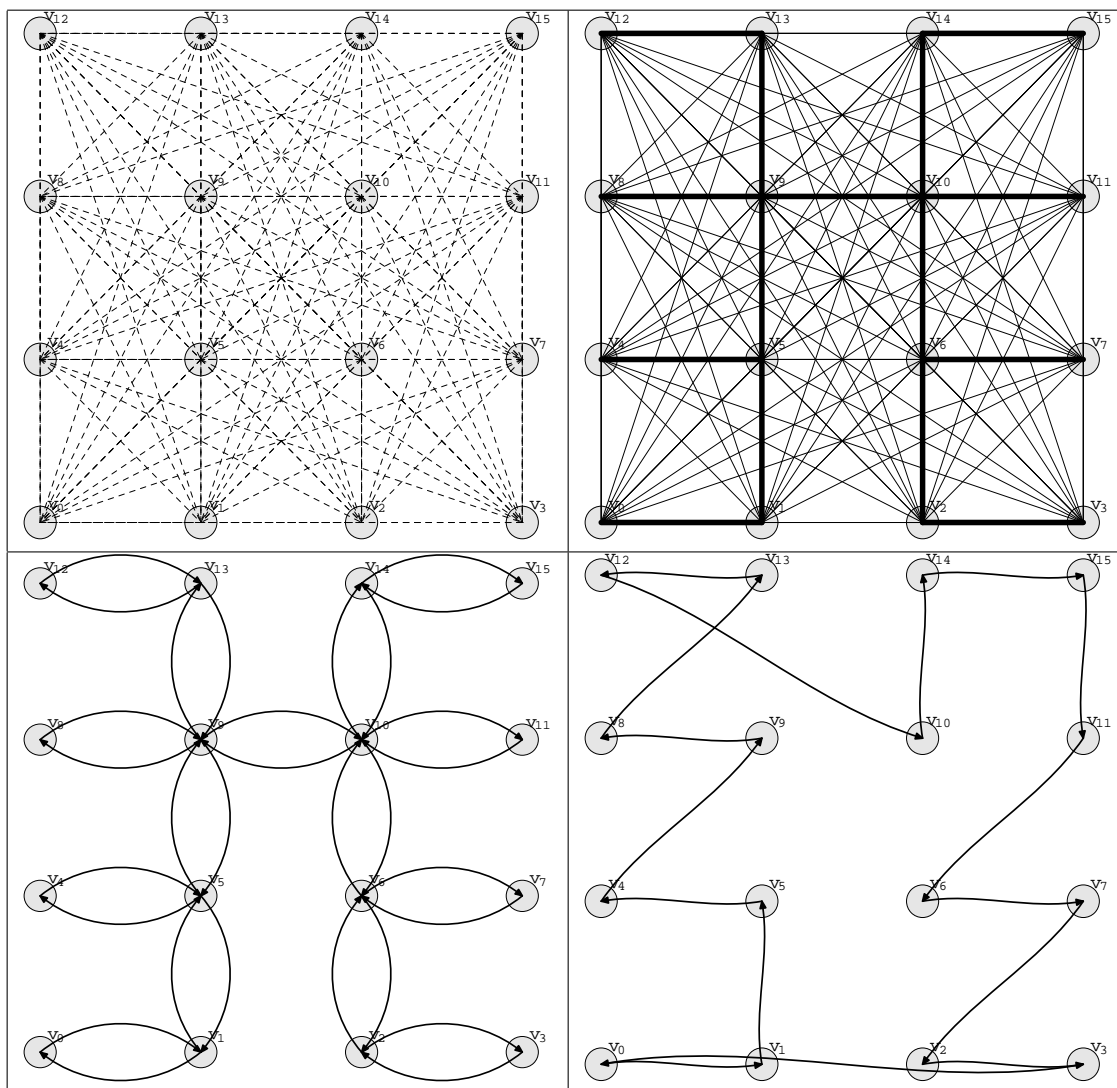
Это и означает, что длина найденного гамильтонова цикла превосходит длину кратчайшего гамильтонова цикла не более, чем в два раза. \square

Однако, лучшим по точности полиномиальным приближенным алгоритмом для метрической задачи коммивояжера остается алгоритм Кристофидеса (алгоритм 13), предложенный им в 1976 г.

Сначала пара определений.

Алгоритм 12 Простой алгоритм для решения метрической задачи коммивояжера

1. Найти минимальное остовное дерево T с матрицей весов $[d_{ij}]$ (пунктиром обозначены ребра не вошедшие в минимальное остовное дерево).
2. Построить эйлеров граф G и эйлеров обход в G , продублировав все ребра дерева T
3. Из эйлерова маршрута (обхода) гамильтонов цикл строится путем последовательного вычеркивания вершин, встретившихся ранее.



Определение 28 Паросочетание — подмножество ребер графа, такое что никакие два ребра из этого подмножества не инцидентны какой-либо одной вершине.

Определение 29 Совершенное паросочетание — паросочетание, покрывающее все вершины графа.

Отметим, что алгоритм 13 является полиномиальным алгоритмом. Шаг 1 может выполнен за время $O(n)$, шаг 2 — за время $O(n^3)$. Шаги 3–4 выполняются за линейное время.

Теорема 14 Отношение длины пути, построенного алгоритмом 13 к длине оптимального пути, не превосходит $3/2$.

Доказательство Во-первых, отметим, что построенный граф G_E действительно эйлеров (согласно теореме 12), т.к. каждая вершина, которая имеет четную степень до добавления ребер паросочетания T , имеет ту же степень, а степень каждой вершины нечетной степени — увеличивается на единицу (из-за добавления ребра паросочетания). Кроме того, граф связан, т.к. содержит остовное дерево.

Длина результирующего гамильтонова цикла P_H удовлетворяет неравенству:

$$c(P_H) \leq c(P_E) = c(G_E) = c(T) + c(M).$$

С другой стороны, для кратчайшего гамильтонова цикла P_H^* выполнено:

$$c(T) \leq c(P_H^*) \leq c(P_H).$$

Пусть i_1, i_2, \dots, i_{2m} — множество вершин нечетной степени в T , в том порядке каком они появляются в P_H^* . Рассмотрим два паросочетания на этих вершинах:

$$\begin{aligned} M_1 &= \{i_1, i_2\}, \{i_3, i_4\}, \dots, \{i_{2m-1}, i_{2m}\} \\ M_2 &= \{i_2, i_3\}, \{i_4, i_5\}, \dots, \{i_{2m}, i_1\}. \end{aligned}$$

Из неравенства треугольника (докажите это в качестве упражнения)

$$c(P_H^*) \geq c(M_1) + c(M_2).$$

Но, M — кратчайшее паросочетание, значит $c(M_1) \geq c(M)$, $c(M_2) \geq c(M)$. Имеем:

$$c(P_H^*) \geq 2c(M) \Rightarrow c(M) \leq \frac{1}{2}c(P_H^*).$$

Окончательно имеем

$$c(P_H) \leq c(T) + c(M) \leq c(P_H^*) + \frac{1}{2}c(P_H^*) = \frac{3}{2}c(P_H^*).$$

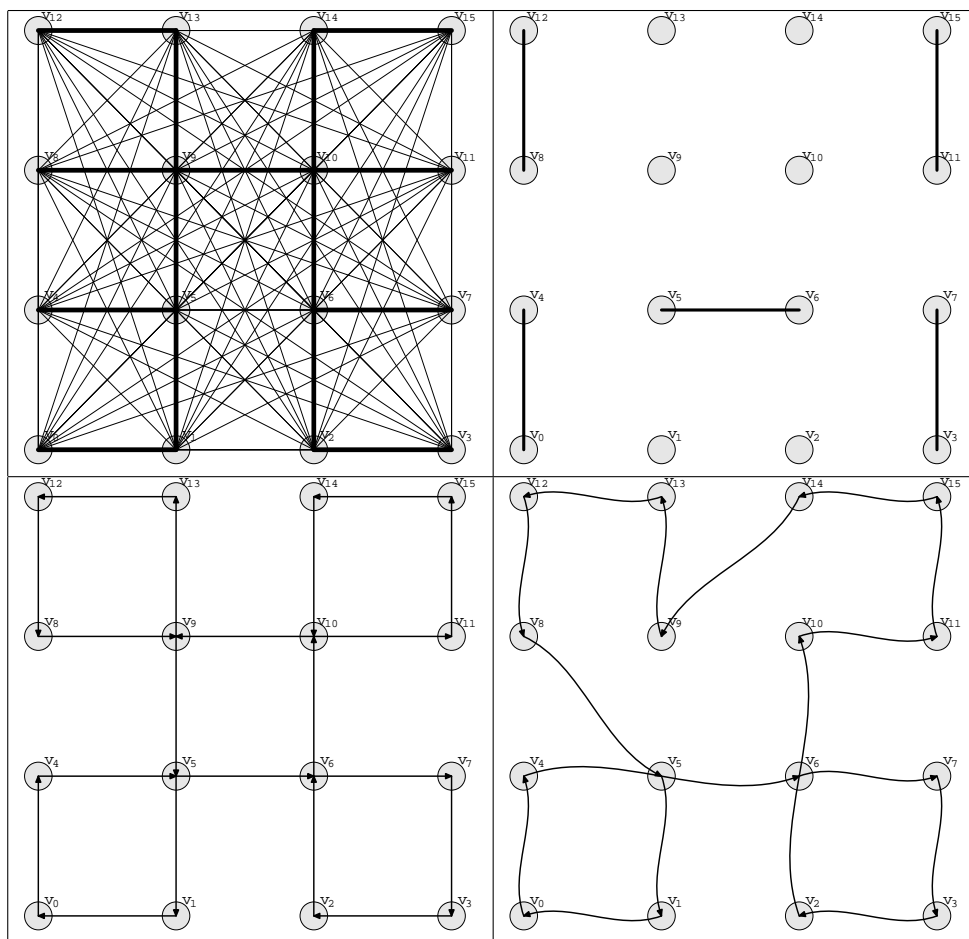
□

Алгоритм 13 Алгоритм Кристофидеса для решения метрической TSP

Вход: $m \times n$ матрица $[d_{ij}]$ для графа G — постановка задачи 2 согласно определению 27.

Выход: гамильтонов цикл P_H для графа G .

1. Найти минимальное остовное дерево T с матрицей весов $[d_{ij}]$;
2. Выделить множество $N(T)$ всех вершин нечетной степени в T и найти кратчайшее совершенное паросочетание M в полном графе G с множеством вершин $N(T)$;
3. Построить эйлеров граф G_E с множеством вершин $\{v_1, \dots, v_n\}$ и множеством ребер $T \cup M$;
4. Найти эйлеров обход P_E в G_E ;
5. Построить гамильтонов цикл (соответствующий вложенный тур) P_H из P_E , последовательным вычеркиванием посещенных вершин.



2.3 Динамическое программирование для задачи о рюкзаке

В силу важности метода динамического программирования для построения эффективных приближенных алгоритмов мы остановимся на нем несколько подробнее и проиллюстрируем его на примере следующей **NP**-полной задачи *сумма размеров*:

Задача 11 СУММА РАЗМЕРОВ

Даны натуральные числа a_1, \dots, a_n , называемые размерами предметов и число A .

Существует ли решение в 0-1 переменных (x_1, \dots, x_n) уравнения $\sum_{i=1}^n a_i x_i = A$?

Обозначим через T множество частичных сумм $\sum_{i=1}^n a_i x_i$ для всех 0–1 векторов (x_1, \dots, x_n) и через $T(k)$ – множество всех частичных сумм $\sum_{i=1}^k a_i x_i$, которые не превосходят A .

Для того чтобы решить задачу достаточно проверить включение $A \in T$. Если оно выполнено, то ответ ДА, в противном случае – НЕТ. Множество всех частичных сумм T может быть построено путем просмотра всех 0-1 векторов ("полным перебором" 2^n векторов). Динамическое программирование дает полиномиальный алгоритм, основанный на рекуррентном соотношении

$$T(k+1) = T(k) \cup (T(k) + a_{k+1}),$$

где $(T(k) + a_{k+1})$ обозначает множество всех чисел вида $s + a_{k+1}$, $s \in T(k)$, которые не превосходят A .

Алгоритм 14 Динамическое программирование для задачи СУММА РАЗМЕРОВ

Вход: $n > 0$, $A > 0$, $a_i > 0$, $\forall i \in (1, \dots, n)$

Выход: Да, если существует решение уравнения $\sum_{i=1}^n a_i x_i = A$ в 0/1 переменных, нет в противном случае.

$T(0) := \{0\}$

for all $i \in \{1..n\}$ **do** {по всем предметам}

$T(i+1) := \emptyset$

for all $x \in T(i)$ **do** {по всем построенным частичным суммам}

if $x + a_{i+1} \leq A$ **then**

$T(i+1) := T(i+1) \cup \{x + a_{i+1}\}$ {добавляем новую частичную сумму}

end if

end for

$T(i+1) := T(i+1) \cup T(i)$

end for

return $(A \in T(n))$

Отметим, что этот алгоритм на самом деле находит все различные значения частичных сумм и, таким образом, дает точное решение задачи. Число шагов алгоритма есть величина $O(nA)$, поскольку:

1. число циклов равно n ;
2. размер каждого множества $T(i)$ не превосходит A .

При небольших значениях A снижение сложности с 2^n до An дает очевидный эффект, и это характерно для многих псевдополиномиальных алгоритмов.

Так как задача СУММА РАЗМЕРОВ **NP**-полна, рассчитывать на построение для нее алгоритма существенно лучшего, чем псевдополиномиальный алгоритм 14 не приходится.

Пример. Пусть $a_1 = 8$, $a_2 = 10$, $a_3 = 7$, $a_4 = 5$ и $A = 19$. Тогда

$$\begin{aligned} T(1) &= \{0, 8\} \\ T(2) &= \{0, 8, 10, 18\} \\ T(3) &= \{0, 8, 10, 18, 7, 15, 17\} \\ T(4) &= \{0, 8, 10, 18, 7, 15, 17, 5, 12, 13\}. \end{aligned}$$

Упражнение 32 Рассмотрим модификацию задачи СУММА РАЗМЕРОВ, разрешим даже отрицательные размеры. Формально: Даны натуральные числа a_i , $\forall i \in [1 \dots n]$ — $n^2 \leq a_i \leq n^2$, и число A . Существует ли решение в 0-1 переменных (x_1, \dots, x_n) уравнения $\sum_{i=1}^n a_i x_i = A$?

Существует ли полиномиальный алгоритм для этой задачи?

Рассмотрим теперь внешне похожую **NP**-полную задачу 0-1 РЮКЗАК и проиллюстрируем как метод динамического программирования работает для нее.

Задача 12 0-1 РЮКЗАК (Knapsack) Даны натуральные числа a_1, \dots, a_n (называемые размерами предметов), c_1, \dots, c_n (стоимости предметов) и B (размер рюкзака).

Найти максимальное значение f^ целевой функции*

$$f \equiv \sum_{i=1}^n c_i x_i \rightarrow \max$$

с ограничением на размер рюкзака

$$\sum_{i=1}^n a_i x_i \leq B, \quad x_i \in \{0, 1\}.$$

2.3. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ ДЛЯ ЗАДАЧИ О РЮКЗАКЕ67

Содержательно задача означает выбор предметов с наибольшей суммарной стоимостью, уместяющихся в рюкзак заданного размера. Эта задача часто возникает при выборе оптимального управления в различных экономико-финансовых областях (распределение бюджета отдела по проектам и т.п.).

Как и прежде, достаточно найти множество $T(k)$ всех пар (c, A) , где c — произвольная сумма $\sum_{i=1}^k c_i x_i$ и A — соответствующая сумма $\sum_{i=1}^k a_i x_i \leq B$.

Описанный выше способ позволяет рекурсивно построить множество $T(n)$ за Bf^*n шагов. Это аналогично заполнению за n -шагов 0/1 таблицы, размером $B \times f^*$, представляющую пространство всевозможных допустимых (не обязательно оптимальных) решений задачи 12.

Однако уменьшить перебор можно учитывая следующий

Ключевой факт. Если на k -ом шаге у нас имеются несколько частичных решений с одинаковой массой, но различной стоимостью, то можно для каждой массы оставить решения лишь с максимальной стоимостью, а если есть несколько частичных решений с одинаковой стоимостью, но различной массой, то можно смело выкинуть решения с большей массой. При любом из упомянутых действий мы не потеряем оптимальное решение!

Упражнение 33 Докажите это.

Соответственно, это означает отсутствие необходимости держать в памяти все частичные решения, достаточно на каждой итерации помнить либо о не больше чем B наиболее "дорогих" частичных решениях, либо о не больше чем f^* наиболее "легких" решениях. Для иллюстрации посмотрим на пространство частичных решений (оптимальное решение показано как \oplus) для следующего набора предметов: (\$2,1кг), (\$3,2кг), (\$4,3кг), (\$6,5кг), (\$2,4кг), (\$5,2кг).

	самые легкие	1кг	2кг	3кг	4кг	5кг
самые дорогие		\$2	\$5	\$7	\$6	\$9
\$1						
\$2	1кг	*			*	
\$3	2кг		*			
\$4	3кг			*		*
\$5	2кг		*	*		
\$6	4кг				*	*
\$7	3кг			*		*
\$8						
\$9	5кг					\oplus
\$10						

Алгоритм 15 помнит о наиболее "легких" частичных решениях, и тем самым дает точное решение задачи за время $O(nf^*)$. Действительно,

Алгоритм 15 Динамическое программирование для задачи 0-1 РЮКЗАК

Вход: $n > 0$, $B > 0$, $c_i > 0$, $b_i > 0$, $\forall i \in (1, \dots, n)$ **Выход:** $\max_{(x_1, \dots, x_n), x_i \in (0,1)} \sum_{i=1}^n c_i x_i$

{Lightest — динамический массив.}

{Lightest(cost) — минимальный вес построенного частичного решения стоимости cost}

undefine(Lightest)

Lightest(0) \leftarrow {0} {Пустое решение тоже имеет свою стоимость}**for all** $i \in \{1..n\}$ **do** {по всем предметам} **for all** $j : \text{Lightest}(j) \text{ is defined}$ **do** {по всем отобранным частичным решениям} $cost \leftarrow j + c_i$ $weight \leftarrow \text{Lightest}(j) + a_i$ **if** $weight \leq B$ and $(\text{Lightest}(cost) \text{ is not defined or } \text{Lightest}(cost) > weight)$ **then** $\text{Lightest}(cost) \leftarrow weight$ {отбираем новое частичное решение} **end if** **end for****end for****return** $\max_{\text{Lightest}(Cost) \text{ is defined}} Cost$

1. число циклов равно n ;
2. размер множества отобранных частичных решений не превосходит f^* ;

Упражнение 34 Постройте алгоритм динамического программирования для задачи 12, основанный на отборе наиболее “дорогих” частичных решений. Какова будет его временная сложность?

Таким образом, для небольших значений параметров c_1, \dots, c_n , или B , можно построить эффективный псевдополиномиальный алгоритм для точного решения задачи о рюкзаке. Еще раз отметим, что он не является полиномиальным. При росте значений параметров c_1, \dots, c_n, B эффективность этого алгоритма будет уменьшаться. Кроме этого, стоит помнить и о пространственной сложности алгоритма — нам требуется $\Omega(B)$ или $\Omega(f^*)$ памяти для хранения частичных решений.

Что же делать, если параметры c_1, \dots, c_n, B задачи велики настолько, что алгоритм 15 не укладывается в требования технического задания по времени или по памяти?

Оказывается, в этом случае можно использовать идею алгоритма 15 для решения задачи 0-1 РЮКЗАК с любой наперед заданной точностью.

2.4 Полностью полиномиальная приближенная схема для задачи о рюкзаке

Одним из общих подходов к решению переборных задач является разработка приближенных алгоритмов с гарантированными оценками качества получаемого решения². Особую роль среди приближенных алгоритмов играют те, которые способны находить решения с любой фиксированной наперед заданной точностью (см. определение 23).

Определение 30 Алгоритм с мультипликативной точностью $(1 + \varepsilon)$ называется ε -оптимальным.

Тот же термин ε -оптимальное используется для обозначения допустимого решения со значением целевой функции, отличающемся от оптимума не более, чем в $(1 + \varepsilon)$ раз (таким образом, задача, стоящая перед ε -оптимальным алгоритмом состоит в отыскании какого-либо ε -оптимального решения).

Определение 31 Полностью полиномиальной аппроксимационной схемой называется приближенный алгоритм, в котором уровень точности ε выступает в качестве нового параметра, и алгоритм находит ε -оптимальное решение за время, ограниченное полиномом от длины входа и величины ε^{-1} .

Только одно обстоятельство является препятствием для построения полностью полиномиальной аппроксимационной схемы для задачи 0-1 РЮКЗАК методом динамического программирования. Это наличие “больших” коэффициентов в целевой функции.

Действительно, как мы видели выше, динамическое программирование дает точный псевдополиномиальный алгоритм для задачи о рюкзаке со сложностью $O(nf^*)$. Если величина f^* не ограничена сверху никаким полиномом (то есть имеются большие коэффициенты), то этот псевдополиномиальный алгоритм не является полиномиальным.

Однако, к счастью, существует общий метод (который условно можно назвать *масштабированием*), позволяющий перейти к задаче с небольшими коэффициентами в целевой функции, оптимум которой не сильно отличается от оптимума исходной задачи.

Зададимся вопросом: что произойдет, если мы отбросим в каждом из коэффициентов c_1, \dots, c_n младшие t бит? Каково будет изменение значения целевой функции после этой операции?

Ясно, что любое допустимое решение “усеченной” задачи является также допустимым решением исходной задачи. Кроме того, абсолютная погрешность не превосходит величины $n2^t$. Если потребовать, чтобы эта величина не превосходила $f^*\varepsilon/2$,

²Алгоритм, не гарантирующий точность решения, однако применяемый на практике из-за хороших практических результатов принято называть эвристикой

то по определению получим, что каждое допустимое решение исходной задачи отличается от решения “усеченной” задачи не более, чем на величину $f^*\varepsilon/2$.

Обозначая оптимум “усеченной” задачи через \tilde{f} , получаем, что

$$\tilde{f} \geq f^* - f^*\varepsilon/2 = (1 - \varepsilon/2)f^* \geq (1 + \varepsilon)^{-1}f^*,$$

т.е. оптимум “усеченной” задачи отличается от оптимума исходной задачи не более, чем в $1 + \varepsilon$ раз. При этом величина f^* для “усеченной” задачи уменьшается не менее, чем в 2^t раз по сравнению с исходной. И таким образом, для отмасштабированной задачи, алгоритм 15 будет работать существенно меньшее время. Однако проблема состоит в том, что в момент масштабирования мы не знаем величины оптимума f^* , и не можем выбрать оптимальный коэффициент “отсечки” t .

Поэтому, важное наблюдение состоит в том, что вместо f^* можно рассматривать другую величину (оценку), которая отличается от f^* не более, чем в заданное число раз и которую мы умеем находить эффективно. Тривиальная аппроксимация

$$nc_{\max} \geq f^* \geq c_{\max},$$

где $c_{\max} = \max_i c_i$ дает пример такой величины (мы считаем, что размеры всех предметов не превосходят размер рюкзака — в противном случае их просто можно исключить из рассмотрения).

Рассмотрим следующий алгоритм 16.

Алгоритм 16 ε -оптимальный алгоритм для задачи 0-1 РЮКЗАК

Вход: $\varepsilon > 0, n > 0, B > 0, c_i > 0, b_i > 0, \forall i \in (1, \dots, n)$

Выход: $(x_1, \dots, x_n), x_i \in (0, 1) : \frac{\sum_{i=1}^n c_i x_i}{f} \geq \frac{1}{1+\varepsilon}$

$t \leftarrow$ максимальное неотрицательное целое, такое что $n2^t \leq (\varepsilon/2)c_{\max}$

for all $i \in \{1..n\}$ **do** {по всем предметам}

$c_i \leftarrow c_i \div 2^t$ {округляем целевые коэффициенты}

end for

Call Алгоритм 15

Полученное решение является ε -оптимальным поскольку в силу выбора t для аддитивной погрешности алгоритма выполнено неравенство

$$n2^t \leq (\varepsilon/2)c_{\max} \leq (\varepsilon/2)f^*.$$

Какова сложность этого алгоритма? Она, как и для алгоритма 15 есть величина $O(n\tilde{f})$. Поскольку для f^* справедливы оценки

$$f^* \leq nc_{\max},$$

а для “усеченной” задачи максимальный коэффициент в целевой функции не превосходит $c_{\max}/2^t$, то следовательно $\tilde{f} \leq n \frac{c_{\max}}{2^t}$. В силу выбора t имеем

$$n2^{t+1} > (\varepsilon/2)c_{\max},$$

2.4. ПОЛНОСТЬЮ ПОЛИНОМИАЛЬНАЯ ПРИБЛИЖЕННАЯ СХЕМА ДЛЯ ЗАДАЧИ О РЮКЗАКЕ

откуда

$$c_{\max}/2^t < 4n/\varepsilon.$$

Это дает оценку сложности

$$n\tilde{f} \leq n^2 c_{\max}/2^t \leq n^2 4n\varepsilon^{-1},$$

и окончательно имеем оценку сложности алгоритма 16 вида $O(n^3\varepsilon^{-1})$.

Можно ли улучшить эту оценку? Ответ на этот вопрос положителен. Для этого рассмотрим менее наивную аппроксимацию величины f^* . Упорядочим по возрастанию элементы массива $c_1/a_1, c_2/a_2, \dots, c_n/a_n$. В соответствии с этим упорядочиванием полагаем переменные x_i равными 1 (типичный *жадный алгоритм*) до тех пор пока выполнено ограничение

$$\sum_{i=1}^n a_i x_i \leq B.$$

Сравним затем стоимость полученного допустимого решения с максимальным коэффициентом c_{\max} и выберем максимум из этих двух чисел. Это так называемый *модифицированный жадный алгоритм*, дающий приближенное допустимое решение, которое мы обозначим через x_G , а значение целевой функции для него — через f_G . Нетрудно доказать, что выполнены следующие неравенства:

Упражнение 35 Для значения решения f_G полученного модифицированным жадным алгоритмом для задачи о рюкзаке, и оптимального значения f^* выполняется

$$f^*/2 \leq f_G \leq f^*. \quad (2.3)$$

Запишем в явном виде полученный алгоритм (См. алгоритм 17), дающий более эффективный способ приближенного решения задачи о рюкзаке с любой наперед заданной точностью ε .

Полученный вектор также является ε -оптимальным решением в силу выбора величины t . Какова сложность этого алгоритма? Как и раньше, сложность есть величина вида $O(n\tilde{f})$. Поскольку для \tilde{f} справедливы оценки (2.3), получаем

$$\tilde{f} \leq 4n\varepsilon^{-1},$$

что дает оценку сложности алгоритма 17 вида $O(n^2\varepsilon^{-1})$.

Алгоритм 17 Улучшенный ε -оптимальный алгоритм для задачи 0-1 РЮКЗАК

Вход: $\varepsilon > 0, n > 0, B > 0, c_i > 0, b_i > 0, \forall i \in (1, \dots, n)$

Выход: $(x_1, \dots, x_n), x_i \in (0, 1) : \frac{\sum_{i=1}^n c_i x_i}{f} \geq \frac{1}{1+\varepsilon}$

Sort($\{\frac{c_i}{a_i}\}_{i=1}^n$) {Сортируем по возрастанию массив $c_1/a_1, c_2/a_2, \dots, c_n/a_n$ }

{мы далее считаем $c_1/a_1 \leq c_2/a_2 \leq \dots \leq c_n/a_n$ }

$f_G \leftarrow 0$

$b_G \leftarrow 0$

for all $i \in \{1..n\}$ **do** {в порядке убывания перспективности предметов}

if $b_G + a_i \leq B$ **then**

 {Добавляем предмет в жадное решение}

$b_G \leftarrow b_G + a_i$

$f_G \leftarrow f_G + c_i$

end if

end for

$t \leftarrow$ максимальное неотрицательное целое, такое что $n2^t \leq (\varepsilon/2)f_G$

for all $i \in \{1..n\}$ **do** {по всем предметам}

$c_i \leftarrow c_i \div 2^t$ {округляем целевые коэффициенты}

end for

Call Алгоритм 15

Глава 3

Анализ сложности в среднем. Вероятностные алгоритмы.

3.1 Анализ сложности в среднем. Задача упаковки.

Среди подходов к решению NP-трудных задач можно выделить два. Первый заключается в построении приближенных алгоритмов с гарантированными оценками точности получаемого решения (См. раздел 2), а второй — в отказе от анализа сложности алгоритмов по наихудшему случаю и переходе к анализу сложности в среднем.

Настоящая лекция посвящена изучению второго подхода (анализу сложности в среднем) применительно к задаче упаковки.

При анализе сложности в среднем вводится некоторая вероятностная мера на исходных данных. Средним временем работы алгоритма называется математическое ожидание времени его работы (по всем исходным данным с заданным распределением).

Рассмотрим этот подход на примере задачи об упаковке.

Задача 13 ЗАДАЧА ОБ УПАКОВКЕ Дано конечное множество L из m элементов и система его подмножеств S_1, \dots, S_n . Требуется найти максимальную по числу подмножеств подсистему попарно непересекающихся подмножеств.

Эта задача эквивалентна следующей 0–1 целочисленной линейной программе¹:

$$\begin{aligned} \mathbf{c}\mathbf{x} &\rightarrow \max \\ A\mathbf{x} &\leq \mathbf{b} \\ \forall i \ x_i &\in \{0, 1\} \end{aligned} \tag{3.1}$$

¹Здесь $\mathbf{x} \leq \mathbf{y}$ обозначает покомпонентное сравнение векторов.

В этой целочисленной линейной программе n столбцов-переменных x_1, \dots, x_n соответствуют подмножествам S_1, \dots, S_n и их включению в решение. $(0, 1)$ -матрица A (размера $m \times n$) описывает состав каждого подмножества (или принадлежность каждого из m предметов, заданным n подмножествам). Проще говоря, матрица A является *матрицей инцидентности* семейства S_1, \dots, S_n подмножеств множества L .

Напомним определения:

Определение 32 Элемент l_i и подмножество S_j **инцидентны**, если $l_i \in S_j$.

Определение 33 Матрицей инцидентности для семейства S_1, \dots, S_n подмножеств множества L называется $(0, 1)$ -матрица $A = (a_{ij})$ размером $m \times n$, построенная по следующему правилу: $a_{ij} = 1$ в том и только в том случае, когда элемент $l_i \in L$ и подмножество S_j инцидентны.

Это позволяет при изучении задач о покрытии/упаковке пользоваться как терминологией подмножеств конечного множества, так и вести рассмотрения в терминах $(0, 1)$ -матриц.

Векторы стоимости и ограничений — состоящие полностью из единиц векторы размерности n и m соответственно. Таким образом m строк-ограничений, соответствующих m предметам, очевидным образом выражают ограничение на непересекаемость выбранных подмножеств ни по одному предмету, а целевая функция равна числу таких подмножеств.

В литературе задача с ЦЛП постановкой 3.1 (с возможностью выбора произвольных неотрицательных целочисленных векторов c и b), также встречается под названием УПАКОВКА ПОДМНОЖЕСТВ.

Известно задача 13 NP-полна [Joh90].

Очевидно, вектор $(0, \dots, 0)$ является допустимым в (3.1). Предположим, что все элементы a_{ij} матрицы ограничений из (3.1) являются независимыми случайными величинами, принимающими значения $\{0, 1\}$, причем для всех i, j выполнено

$$P\{a_{ij} = 1\} = p, \quad P\{a_{ij} = 0\} = 1 - p.$$

При этом время работы алгоритма является случайной величиной, зависящей от исходных данных, и под средним временем понимается математическое ожидание времени работы алгоритма.

Определение 34 Алгоритм называется **полиномиальным в среднем**, если математическое ожидание времени его работы ограничено сверху некоторым полиномом от длины входа.

Теорема 15 Пусть выполнено условие

$$mp^2 \geq \ln n \quad (3.2)$$

Тогда имеется модификация метода динамического программирования для задачи (13), являющаяся полиномиальным в среднем алгоритмом.

Основу метода динамического программирования применительно к задаче УПАКОВКИ составляют рекуррентные соотношения, позволяющие без полного перебора построить все допустимые решения задачи. Опишем один из вариантов этого метода.

Обозначим через $T(k)$ множество всех допустимых булевых векторов для системы (3.1) с $n-k$ нулевыми последними компонентами и через e_k - вектор размерности n с единичной k -й компонентой и остальными нулевыми компонентами. Рассмотрим алгоритм 18.

Алгоритм 18 Полиномиальный в среднем алгоритм для задачи об упаковке

Вход: $m, n > 0$, $A \in (0, 1)$ матрица размера $m \times n$

Выход: (x_1, \dots, x_n) — решение (3.1)

$T(0) := \{(0, \dots, 0)\}$

for all $j \in \{1..n\}$ **do** {по всем подмножествам}

$T(j) := \emptyset$

for all $x \in T(j)$ **do** {по всем построенным допустимым решениям}

if $x + e_k$ допустимый для (3.1) **then**

$T(j) := T(j) \cup \{x + e_k\}$ {добавляем новое допустимое решение}

end if

end for

$T(j) := T(j) \cup T(j-1)$

end for

return $x : x \in T(n), \sum_{i=1}^n x_i \rightarrow \max$

Нетрудно убедиться, что сложность алгоритма В есть $O(mn|T(n)|)$. Действительно — внешний цикл $O(n)$, внутренний $O(|T(n)|)$, проверка на допустимость нового решения — $O(m)$ (если непонятно, обязательно подумайте почему).

Математическое ожидание времени работы алгоритма — $O(nm\mathbf{E}|T(n)|)$. Поэтому для доказательства теоремы 15 достаточно будет оценить сверху математическое ожидание размера множества $T(n)$.

Вероятность $P(k)$ того, что некоторый вектор \mathbf{x}^k с k единичными компонентами и $n-k$ нулевыми компонентами является допустимым решением (3.1) (то есть удовлетворяет всем ограничениям в (3.1)) можно оценить сверху следующим образом:

$$P(k) = \prod_{i=1}^m p_i(k),$$

где $p_i(k)$ — вероятность выполнения i -неравенства для \mathbf{x}^k :

$$p_i(k) \equiv P \left\{ \sum_{j=1}^n a_{ij} x_j^k \leq 1 \right\} \leq P \left\{ \sum_{j=1}^k a_{ij} \leq 1 \right\},$$

откуда (при $k > 1$)

$$\begin{aligned} P(k) &\leq \prod_{i=1}^m P \left\{ \sum_{j=1}^k a_{ij} \leq 1 \right\} = \prod_{i=1}^m \left(P \left\{ \sum_{j=1}^k a_{ij} = 0 \right\} + P \left\{ \sum_{j=1}^k a_{ij} = 1 \right\} \right) \\ &= \prod_{i=1}^m ((1-p)^k + kp(1-p)^{k-1}) \\ &= \prod_{i=1}^m (1-p)^{k-1} (1+p(k-1)) \leq \prod_{i=1}^m (1-p)^{k-1} (1+p)^{k-1} \\ &= \prod_{i=1}^m (1-p^2)^{k-1} \leq \prod_{i=1}^m e^{-p^2(k-1)} = e^{-mp^2(k-1)} \end{aligned}$$

Следовательно, мы можем оценить математическое ожидание мощности $T(n)$ следующим образом

$$\begin{aligned} \mathbf{E}|T(n)| &\leq \sum_{k=0}^n \binom{n}{k} P(k) = 1 + \sum_{k=1}^n \binom{n}{k} P(k) = 1 + n + \sum_{k=2}^n \binom{n}{k} P(k) \\ &= 1 + n + \sum_{k=2}^n \binom{n}{k} e^{-mp^2(k-1)} \leq 1 + n + \sum_{k=2}^n e^{k \ln n - mp^2(k-1)} \\ &= 1 + n + n \sum_{k=2}^n e^{(k-1)(\ln n - mp^2)} \end{aligned}$$

При условии $mp^2 > \ln n$ в последней сумме каждый член не превосходит 1. Это и означает, что $\mathbf{E}|T(n)| = O(n^2)$.

В заключение отметим, что из полученных результатов вытекает не только полиномиальный в среднем алгоритм для задачи оптимизации (3.1), но и полиномиальный в среднем алгоритм для, вообще говоря, более трудной задачи подсчета числа целых точек в многограннике (3.1).

3.2 Точность жадного алгоритма для почти всех исходных данных

Анализ точности алгоритмов в типичном случае. Асимптотическая точность жадного алгоритма в задаче о покрытии для "почти всех исходных данных"

3.2. ТОЧНОСТЬ ЖАДНОГО АЛГОРИТМА ДЛЯ ПОЧТИ ВСЕХ ИСХОДНЫХ ДАННЫХ 77

Ранее мы доказали, что для задачи о покрытии жадный алгоритм в худшем случае гарантирует нахождение покрытия, размер которого превосходит размер минимального покрытия не более чем в логарифмическое число раз (по m).

Наша цель сейчас — показать, что для "типичных данных" (в отличие от худшего случая) это отношение близко к единице, т.е. размер покрытия, найденного жадным алгоритмом, почти равен размеру минимального покрытия.

Как и в предыдущем разделе, мы будем использовать формулировку задачи о покрытии на языке $(0, 1)$ -матриц и целочисленных линейных программ:

$$\begin{aligned} \mathbf{c}\mathbf{x} &\rightarrow \min \\ \mathbf{A}\mathbf{x} &\geq \mathbf{b} \\ \forall i \ x_i &\in \{0, 1\} \end{aligned} \tag{3.3}$$

В этой целочисленной линейной программе n столбцов-переменных x_1, \dots, x_n соответствуют подмножествам S_1, \dots, S_n и их включению в решение-покрытие. Матрица A также является матрицей инцидентности, а векторы стоимости и ограничений — также векторы размерности n и m соответственно, состоящие полностью из единиц.

Таким образом, m строк-ограничений, соответствующих m предметам, очевидным образом выражают ограничение на обязательное включение предмета хотя бы в одно из покрывающих подмножеств, а целевая функция равна числу таких подмножеств.

Как и в предыдущем разделе, предположим, что $A = (a_{ij})$ — случайная $(0, 1)$ -матрица, такая, что для всех i, j выполнено

$$P\{a_{ij} = 1\} = p, \quad P\{a_{ij} = 0\} = 1 - p.$$

Пусть $R(G) = \frac{Z_G}{M}$, где Z_G — величина покрытия найденного жадным алгоритмом, а M — величина минимального покрытия. Наш основной результат заключается в следующей теореме.

Теорема 16 Пусть вероятность p фиксирована, $0 < p < 1$. Пусть для задачи (3.3) со случайной матрицей A определенной выше выполнены условия:

$$\frac{\ln \ln n}{\ln m} \rightarrow 0 \text{ при } n \rightarrow \infty, \tag{3.4}$$

$$\frac{\ln m}{n} \rightarrow 0 \text{ при } n \rightarrow \infty. \tag{3.5}$$

Тогда для любого фиксированного $\varepsilon > 0$

$$\mathbf{P}\{R(G) \leq 1 + \varepsilon\} \rightarrow 1 \text{ при } n \rightarrow \infty.$$

Заметим, что при $m = cn$, где c — некоторая константа, условия (3.4) и (3.5) теоремы 16 выполнены автоматически.

Доказательство Сначала докажем нижнюю оценку размера минимального покрытия.

Пусть X — случайная величина, равная числу покрытий размера

$$l_0 = -\lceil (1 - \delta) \ln m / \ln(1 - p) \rceil,$$

тогда

$$\mathbf{E}X = \binom{n}{l_0} P(l_0),$$

где $P(l_0)$ — вероятность того, что фиксированные l_0 столбцов являются покрытием A . Нетрудно видеть (убедитесь, что это действительно нетрудно!), что

$$P(l_0) = (1 - (1 - p)^{l_0})^m \leq \exp\{-m(1 - p)^{l_0}\}.$$

Используя неравенство $\binom{n}{k} \leq n^k$, получаем

$$\begin{aligned} \ln \mathbf{E}X &\leq l_0 \ln n - m(1 - p)^{l_0} \\ &\leq -\frac{\ln m}{\ln(1 - p)} \ln n - m \exp\left\{-(1 - \delta) \ln(1 - p) \frac{\ln m}{\ln(1 - p)}\right\} \\ &\leq -\frac{\ln m}{\ln(1 - p)} \ln n - mm^{-1}m^\delta \\ &\leq -\frac{\ln m}{\ln(1 - p)} \ln n - m^\delta. \end{aligned}$$

Нетрудно проверить, что для любого фиксированного $0 < \delta < 1$ при условиях (3.4) последнее выражение стремится к $-\infty$ при n стремящемся к бесконечности.

Таким образом, вероятность того, что нет покрытия размера l_0 в случайной $(0, 1)$ -матрице A стремится к 1, поскольку (согласно неравенству Чебышева)

$$\mathbf{P}\{X \geq 1\} \leq \mathbf{E}X \rightarrow 0.$$

Последнее означает, что $M \geq l_0$.

Теперь докажем верхнюю оценку размера покрытия построенного жадным алгоритмом.

Используем следующую известную лемму [AS92] о вероятности больших отклонений для сумм независимых случайных величин.

Лемма 1 Пусть Y — сумма n независимых случайных величин, каждая из которых принимает значение 1 с вероятностью p и 0 — с вероятностью $1 - p$. Тогда

$$\mathbf{P}\{|Y - np| > \delta np\} \leq 2 \exp\{-(\delta^2/3)np\}.$$

3.2. ТОЧНОСТЬ ЖАДНОГО АЛГОРИТМА ДЛЯ ПОЧТИ ВСЕХ ИСХОДНЫХ ДАННЫХ 79

По этой лемме вероятность того, что некоторый фиксированный столбец содержит менее чем $(1 - \delta)pn$ единиц (или более чем $(1 + \delta)pn$ единиц) оценивается сверху так:

$$P_{bad} \leq 2 \exp\{-(\delta^2/3)np\},$$

и математическое ожидание числа таких столбцов не превосходит mP_{bad} . Нетрудно видеть, что

$$mP_{bad} \leq 2 \exp\{\ln m - (\delta^2/3)np\} = 2 \exp\{\ln m - O(1)n\} \rightarrow 0, \text{ при } n \rightarrow \infty,$$

по условию (3.5). Следовательно, первое неравенство Чебышева $P\{X \geq 1\} \leq \mathbf{E}X$ влечет, что вероятность события "каждый столбец содержит не менее $(1 - \delta)pn$ единиц" стремится к 1.

Теперь мы можем почти дословно повторить получение верхней оценки размера покрытия, построенного жадным алгоритмом, в худшем случае.

Пусть N_t — число непокрытых строк после t -го шага жадного алгоритма. Имеем:

$$\begin{aligned} N_t &\leq N_{t-1} - \frac{N_{t-1}(1 - \delta)pn}{n} = N_{t-1} (1 - (1 - \delta)p) \\ &\leq N_0 (1 - (1 - \delta)p)^t = m(1 - (1 - \delta)p)^t. \end{aligned}$$

Найдем максимальное t при котором еще есть непокрытый элемент:

$$m(1 - (1 - \delta)p)^t \geq 1$$

Получим:

$$\ln m + t \ln(1 - (1 - \delta)p) \geq 0,$$

или

$$t \geq -\frac{\ln m}{\ln(1 - (1 - \delta)p)}$$

Отсюда получается верхняя оценка мощности "жадного покрытия" в типичном случае:

$$Z_G \leq 1 + \frac{\ln m}{\ln \frac{1}{1 - p(1 - \delta)}}.$$

Упражнение 36 Докажите, что комбинируя это неравенство с нижней оценкой, можно получить что для любого фиксированного $\varepsilon > 0$ и достаточно больших n

$$R(G) \leq 1 + \varepsilon.$$

□

3.3 Вероятностные алгоритмы (Лас-Вегас и Монте-Карло).

Вероятностный алгоритм — это, неформально, алгоритм, исполнение которого (и, соответственно, результат) зависит от случайных величин (или, как часто говорят, от результатов "подбрасывания монеты").

Формальное определение вероятностного алгоритма обобщает классическое определение алгоритма через машину Тьюринга и использует понятие вероятностной машины Тьюринга (ВМТ). В отличие от классической машины Тьюринга в ВМТ имеются состояния, из которых возможен переход в несколько (более одного) состояний. Выбор состояния, куда ВМТ делает переход, определяется результатом некоторого случайного процесса ("подбрасывания монеты"). При этом обычно считают, что вероятности выпадения одинаковы для обеих сторон монеты, а результат подбрасывания отождествляется с числом 0 или 1.

На формальном уровне вероятностный алгоритм — это некоторая вероятностная машина Тьюринга.

Поскольку результат работы вероятностного алгоритма определяется не только входными данными, а также и последовательностью случайных величин (битов), то можно говорить о вероятности того или иного ответа на заданном входе. Необходимо подчеркнуть, что на одних и тех же исходных данных в разных запусках вероятностного алгоритма могут получаться разные результаты.

Особенностью вероятностных алгоритмов является, то что они не всегда дают правильный ответ (могут ошибаться или не давать ответа). По типу возможных ошибок различают вероятностные алгоритмы типа Монте-Карло и вероятностные алгоритмы типа Лас-Вегас. Отличие состоит в следующем. Если Лас-Вегас алгоритм дает ответ, то он всегда правильный, однако в некоторых случаях алгоритм может не дать ответа. Алгоритмы типа Монте-Карло могут давать и неправильные ответы, при этом различают алгоритмы с односторонними и двусторонними ошибками (например, ответ 'да' — всегда правильный, а ответ 'нет' может быть и неправильным).

По аналогии с детерминированными алгоритмами, где эффективные алгоритмы отождествляются с так называемыми полиномиальными алгоритмами, а класс языков (предикатов) распознаваемых в полиномиальное время обозначается P , предикаты распознаваемые эффективными вероятностными алгоритмами образуют классы BPP и RP для Монте-Карло алгоритмов (соответственно, с двусторонними и односторонними ошибками) и ZPP для Лас-Вегас алгоритмов.

Приведем некоторые формальные определения.

Определение 35 Предикат L принадлежит классу BPP , если существуют такие вероятностная машина Тьюринга M и полином $p(n)$, что машина M на входе x остановится за время, не превосходящее $p(|x|)$, причем

1. $L(x) = 1 \Rightarrow M$ с вероятностью большей $2/3$ дает ответ "да"
2. $L(x) = 0 \Rightarrow M$ с вероятностью большей $2/3$ дает ответ "нет"

Предикаты из класса BPP можно считать реально вычислимыми, т.к. вероятностные машины вполне могут рассматриваться как реальные устройства. В этом их основное отличие от недетерминированных алгоритмов.

Другое (эквивалентное) определение класса BPP не использует понятие вероятностной машины Тьюринга.

Определение 36 Предикат L принадлежит классу BPP , если существуют такие предикат $R(x, y) \in P$ и полином $p(n)$, что $|y| \leq p(|x|)$, и

1. $L(x) = 1 \Rightarrow$ доля строк y , для которых $R(x, y) = 1$, больше $2/3$
2. $L(x) = 0 \Rightarrow$ доля строк y , для которых $R(x, y) = 0$, больше $2/3$

Нетрудно доказать, что если в определении класса BPP число $2/3$ заменить на любое фиксированное число, большее $1/2$, то класс BPP не изменится. Следует отметить также, что сделав достаточное число запусков данного вероятностного алгоритма с одними и теми же исходными данными, можно добиться сколь угодно малой вероятности ошибки выбрав ответ (1 или 0), который дало большинство [MR95].

Аналогично определяется класс RP .

Определение 37 Предикат L принадлежит классу RP , если существуют такие предикат $R(x, y) \in P$ и полином $p(n)$, что $|y| \leq p(|x|)$, и

1. $L(x) = 1 \Rightarrow$ доля строк y , для которых $R(x, y) = 1$ больше $1/2$
2. $L(x) = 0 \Rightarrow$ доля строк y , для которых $R(x, y) = 0$ равна 1

Класс ZPP определяется так:

Определение 38 $ZPP = RP \cap co - RP$, где $co - RP = \{L \mid \bar{L} \in RP\}$.

Один из первых примеров вероятностных алгоритмов, более эффективных чем детерминированные, был предложен Фрейвалдом (см. [MR95]) для задачи проверки матричного равенства $AB = C$. Обычный детерминированный алгоритм заключается в перемножении матриц A и B и сравнении результата с C .

Сложность такого алгоритма есть $O(n^3)$ при использовании стандартного алгоритма умножения матриц размера $n \times n$, и $O(n^{2.376})$ при использовании лучшего из известных быстрых алгоритмов матричного умножения.

Вероятностный алгоритм Фрейвалда для этой задачи имеет сложность $O(n^2)$ и заключается в умножении левой и правой частей на случайный булев вектор $\mathbf{x} = (x_1, \dots, x_n)$ с последующим сравнением полученных векторов. Алгоритм выдает ответ, что $AB = C$, если $AB\mathbf{x} = C\mathbf{x}$ и является алгоритмом типа Монте-Карло с односторонней ошибкой [MR95]. При этом $AB\mathbf{x}$ вычисляется как $A(B\mathbf{x})$, что и обеспечивает оценку сложности алгоритма $O(n^2)$.

Корректность алгоритма обеспечивается следующей теоремой.

Теорема 17 Пусть A , B , и C — $n \times n$ матрицы над полем \mathbf{F}^2 , причем $AB \neq C$. Тогда для случайного вектора x выбранного случайно и равномерно из $\{0, 1\}^n$,

$$P\{ABx = Cx\} \leq 1/2.$$

Доказательство Пусть $D = AB - C$. Мы знаем, что D — не полностью нулевая матрица и хотим оценить вероятность того, что $D\mathbf{x} = 0$. Без ограничения общности можно считать, что ненулевые элементы имеются в первой строке и они располагаются перед нулевыми. Пусть \mathbf{d} — вектор, равный первой строке матрицы D , и предположим, что первые k элементов в \mathbf{d} — ненулевые.

$$\mathbf{P}\{D\mathbf{x} = 0\} \leq \mathbf{P}\{\mathbf{d}^T \mathbf{x} = 0\}.$$

Но, $\mathbf{d}^T \mathbf{x} = 0$ тогда и только тогда, когда

$$x_1 = \frac{-\sum_{i=2}^k d_i x_i}{d_1}$$

Для каждого выбора x_2, \dots, x_k правая часть этого равенства фиксирована и равна некоторому $v > 0$. Вероятность, что x_1 равно v , не превосходит $1/2$, поскольку x_1 равномерно распределено на двухэлементном множестве $(\{0, 1\})$.

□

Классическим примером задачи, где вероятностные алгоритмы успешно применяются, является задача проверки тождеств для многочлена от многих переменных:

Задача 14 Верно ли для заданного полинома $P(x_1, \dots, x_n)$, что $P(x_1, \dots, x_n) \equiv 0$?

Следующая лемма (См. [MR95]) по сути описывает вероятностный Монте-Карло алгоритм с односторонней ошибкой.

Лемма 2 Пусть $Q(x_1, \dots, x_n)$ — многочлен от многих переменных степени d над полем F и пусть $S \subseteq F$ — произвольное подмножество. Если r_1, \dots, r_n выбраны случайно, независимо и равномерно из S , то

²элементы матрицы $\in F$

$$\mathbf{P}(Q(r_1, \dots, r_n) = 0 \mid Q(x_1, \dots, x_n) \neq 0) \leq \frac{d}{|\mathbf{S}|}.$$

Доказательство По индукции по n .

Базисный случай $n = 1$ включает полиномы от одной переменной $Q(x_1)$ степени d . Поскольку каждый такой полином имеет не более d корней, вероятность, что $Q(r_1) = 0$ не превосходит $\frac{d}{|\mathbf{S}|}$.

Пусть теперь предположение индукции верно для всех полиномов, зависящих от не более $n - 1$ переменной.

Рассмотрим полином $Q(x_1, \dots, x_n)$ и разложим его по переменной x_1

$$Q(x_1, \dots, x_n) = \sum_{i=0}^k x_1^i Q_i(x_2, \dots, x_n),$$

где $k \leq d$ — наибольшая степень x_1 в Q .

Предполагая, что Q зависит от x_1 , имеем $k > 0$, и коэффициент при x_1^k , $Q_k(x_2, \dots, x_n)$ не равен тождественно нулю. Рассмотрим две возможности.

Первая — $Q_k(r_2, \dots, r_n) = 0$. Заметим, что степень Q_k не превосходит $d - k$, и по предположению индукции вероятность этого события не превосходит $\frac{(d-k)}{|\mathbf{S}|}$.

Вторая — $Q_k(r_2, \dots, r_n) \neq 0$. Рассмотрим следующий полином от одной переменной:

$$q(x_1) = Q(x_1, r_2, r_3, \dots, r_n) = \sum_{i=0}^k Q_i(r_2, \dots, r_n) x_1^i.$$

Полином $q(x_1)$ имеет степень k , и не равен тождественно нулю, поскольку коэффициент при x_1^k есть $Q_k(r_2, \dots, r_n)$. Базовый случай индукции дает, что вероятность события $q(r_1) = Q(r_1, r_2, \dots, r_n) = 0$ не превосходит $\frac{k}{|\mathbf{S}|}$.

Мы доказали два неравенства:

$$\mathbf{P}\{Q_k(r_2, \dots, r_n) = 0\} \leq \frac{d - k}{|\mathbf{S}|};$$

$$\mathbf{P}\{Q(r_1, r_2, \dots, r_n) = 0 \mid Q_k(r_2, \dots, r_n) \neq 0\} \leq \frac{k}{|\mathbf{S}|}.$$

Используя результат следующего упражнения: покажите, что для любых двух событий E_1, E_2 ,

$$\mathbf{P}\{E_1\} \leq \mathbf{P}\{E_1 \mid \overline{E_2}\} + \mathbf{P}\{E_2\},$$

мы получаем, что вероятность события $Q(r_1, r_2, \dots, r_n) = 0$ не превосходит суммы двух вероятностей $(d - k)/|\mathbf{S}|$ и $k/|\mathbf{S}|$, что дает в сумме желаемое $d/|\mathbf{S}|$. \square

Упражнение 37 Имеется квадратная, $n \times n$ матрица A , элементами которой являются линейные функции $f_{ij}(x) = a_{ij}x + b_{ij}$.

Придумайте Монте-Карло алгоритм с односторонней ошибкой, для проверки этой матрицы на вырожденность ($\det A \equiv 0$).

Еще одним примером успеха вероятностных алгоритмов является разработка эффективных вероятностных алгоритмов проверки простоты числа: для натурального числа n заданного в двоичной системе, определить является ли оно простым [SV77]. Только в 2002 г. для этой задачи удалось построить полиномиальный детерминированный алгоритм.

Одной из многочисленных областей, где широко применяются вероятностные алгоритмы, являются параллельные вычисления. Эффективным параллельным алгоритмом (или NC-алгоритмом) называется алгоритм, который на многопроцессорной RAM (так называемой PRAM) с числом процессоров не превосходящих некоторого полинома завершает работу за время ограниченное полиномом от логарифма длины входа. Построение эффективного детерминированного параллельного алгоритма (NC-алгоритма) для нахождения максимального паросочетания в двудольном графе является одной из основных открытых проблем в теории параллельных алгоритмов. Удалось, однако, построить эффективный параллельный вероятностный алгоритм нахождения максимального паросочетания в двудольном графе (так называемый RNC-алгоритм) [MR95].

Интересным свойством предикатов из класса BPP является наличие для них схем из функциональных элементов (boolean circuits) полиномиального размера. Класс всех предикатов, для которых существуют такие схемы обозначается $P/poly$.

Лемма [Adl78]. $BPP \subseteq P/poly$.

3.4 Вероятностное округление

Вероятностное округление нецелочисленного решения до целочисленного. Приближенные вероятностные алгоритмы для задачи MAX-SAT.

Рассмотрим следующую задачу

Задача 15 МАКСИМАЛЬНАЯ ВЫПОЛНИМОСТЬ³

Даны m скобок конъюнктивной нормальной формы (КНФ) с n переменными. Найти значения переменных, максимизирующее число выполненных скобок.

В качестве примера рассмотрим следующую КНФ:

$$(x_1 \vee \bar{x}_2)(x_1 \vee x_2 \vee \bar{x}_3)(x_2 \vee \bar{x}_1)(x_3 \vee \bar{x}_2).$$

³В англоязычной литературе MAX-SAT

Переменные или их отрицания, входящие в скобку, называются литералами (так в первой скобке литералами являются x_1 и \bar{x}_2).

Задача MAX-SAT является NP-трудной, но мы рассмотрим сейчас простое вероятностное доказательство того, что для любых m скобок существуют значения переменных, при которых выполнено не менее $m/2$ скобок.

Теорема 18 *Для любых m скобок существуют значения переменных, при которых выполнено не менее $m/2$ скобок.*

Доказательство Предположим, что каждой переменной приписаны значения 0 или 1 независимо и равновероятно. Для $1 \leq i \leq m$ пусть $Z_i = 1$ если i -я скобка выполнена, и 0 в противном случае.

Для каждой дизъюнкции (скобки) с k литералами (переменными или их отрицаниями) вероятность что эта дизъюнкция не равна 1 при случайном приписывании значений переменным равна 2^{-k} , поскольку это событие имеет место когда значение каждого литерала в дизъюнкции равно 0, а значения разным переменным приписываются независимо. Значит, вероятность что скобка равна 1 есть $1 - 2^{-k} \geq 1/2$ и мат. ожидание $\mathbf{E}Z_i \geq 1/2$. Отсюда, мат. ожидание числа выполненных скобок (равных 1) равно $\sum_{i=1}^m \mathbf{E}Z_i \geq m/2$. Это означает, что есть приписывание значений переменным, при котором $\sum_{i=1}^m Z_i \geq m/2$. \square

Эта теорема дает по существу приближенный вероятностный алгоритм.

Определение 39 *Вероятностный приближенный алгоритм A гарантирует точность D , если для всех входов I*

$$\frac{\mathbf{E}m_A(I)}{m_0(I)} \geq D,$$

где $m_0(I)$ — оптимум, $m_A(I)$ — значение, найденное алгоритмом и решается задача максимизации. A .

Отличие от детерминированных приближенных алгоритмов состоит в рассмотрении мат. ожидания.

Описанный в теореме 18 вероятностный алгоритм дает точность $1/2$ для MAX-SAT. Мы опишем сейчас другой вероятностный алгоритм, гарантирующий точность $3/4$ для этой задачи.

Для этого мы переформулируем MAX-SAT в задачу целочисленного линейного программирования (ЦЛП). Каждой скобке (элементарной дизъюнкции) C_j поставим в соответствие булеву переменную $z_j \in \{0, 1\}$, которая равна 1, если скобка C_j выполнена, каждой входной переменной x_i сопоставляем индикаторную переменную y_i , которая равна 1 если $x_i = 1$ и равна 0 в противном случае. Обозначим C_j^+ индексы переменных в скобке C_j , которые входят в нее без отрицания, а через C_j^- — множество индексов переменных, которые входят в скобку с отрицанием.

Тогда MAX-SAT допускает следующую формулировку в виде задачи ЦЛП:

$$\begin{aligned} \sum_{j=1}^m z_j &\rightarrow \max \\ \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) &\geq z_j, \quad \forall j. \\ y_i, z_j &\in \{0, 1\}, \quad \forall i, j \end{aligned} \tag{3.6}$$

Упражнение 38 Докажите эквивалентность задачи MAX-SAT (задача 15) и ЦЛП (3.6).

Рассмотрим и решим линейную релаксацию целочисленной программы (3.6).

$$\begin{aligned} \sum_{j=1}^m z_j &\rightarrow \max \\ \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) &\geq z_j, \quad \forall j. \\ y_i, z_j &\in [0, 1], \quad \forall i, j \end{aligned} \tag{3.7}$$

Пусть \hat{y}_i, \hat{z}_j — решение линейной релаксации (3.7). Ясно, что $\sum_{j=1}^m \hat{z}_j$ является верхней оценкой числа выполненных скобок для данной КНФ.

Рассмотрим теперь вероятностный алгоритм 19 (так называемое вероятностное округление), более интересный чем простое округление каждой переменной в 1 и 0 с одинаковыми вероятностями. При вероятностном округлении в алгоритме 19 каждая переменная y_i независимо принимает значение 1 с вероятностью \hat{y}_i (и 0 с вероятностью $1 - \hat{y}_i$).

Алгоритм 19 Приближенный вероятностный алгоритм для задачи 15 (MAX-SAT) на основе линейной релаксации

Вход: Формулировка задачи 15 в виде (3.6)

Выход: (y_1, \dots, y_m) — приближенное решение (3.6), со средней точностью $(1 - 1/e)$.

$\hat{y} \leftarrow$ решения линейной релаксации (3.7)

for all $i \in \{1..m\}$ **do**

$y_i \leftarrow 0$

if $\text{random}(0..1) \leq \hat{y}_i$ **then**

$y_i \leftarrow 1$ { $y_i \leftarrow 1$ с вероятностью \hat{y}_i }

end if

end for

return y

Для целого k положим $\beta_k = 1 - (1 - 1/k)^k$.

Лемма 3 Пусть в скобке C_j имеется k литералов. Вероятность, что она выполнена при вероятностном округлении не менее $\beta_k \hat{z}_j$.

Доказательство Поскольку мы рассматриваем отдельно взятую скобку, без ограничения общности можно предположить, что все переменные входят в нее без отрицаний (докажите этот факт в качестве упражнения!). Пусть эта скобка имеет вид: $x_1 \vee \dots \vee x_k$. Из ограничений линейной релаксации (3.7) следует, что

$$\hat{y}_1 + \dots + \hat{y}_k \geq \hat{z}_j.$$

Скобка C_j остается невыполненной при вероятностном округлении, только если каждая из переменных y_i округляется в 0. Поскольку каждая переменная округляется независимо, это происходит с вероятностью $\prod_{i=1}^k (1 - \hat{y}_i)$. Остается только показать, что

$$1 - \prod_{i=1}^k (1 - \hat{y}_i) \geq \beta_k \hat{z}_j.$$

Выражение в левой части достигает минимума при $\hat{y}_i = \hat{z}_j/k$ для всех i . Следовательно, остается показать, что $1 - (1 - z/k)^k \geq \beta_k z$ для всех положительных целых k и $0 \leq z \leq 1$. Поскольку $f(x) = 1 - (1 - x/k)^k$ — вогнутая функция, для доказательства того, что она не меньше линейной функции на отрезке, достаточно проверить неравенство на концах этого отрезка, т.е. в точках $x = 0$ и $x = 1$ (проделайте это в качестве упражнения!). \square

Используя тот факт, что $\beta_k \geq 1 - 1/e$ для всех положительных целых k получаем, что справедлива следующая

Теорема 19 Для произвольного входа задачи MAX-SAT (произвольной КНФ) среднее число скобок, выполненных при вероятностном округлении, не меньше $(1 - 1/e)$ от максимально возможного числа выполненных скобок.

А теперь мы опишем простую, но общую идею, которая позволит получить приближенный вероятностный алгоритм, имеющий точность $3/4$.

А идея такова: на данном входе запускаем два алгоритма и выбираем из решений лучшее. В качестве двух алгоритмов рассматриваем два алгоритма описанных выше:

1. округление каждой переменной независимо в 0 или 1 с вероятностью $1/2$;
2. вероятностное округление решения линейной релаксации соответствующей целочисленной программы (алгоритм 19).

Пусть n_1 обозначает мат.ожидание числа выполненных скобок для первого алгоритма, и n_2 — мат. ожидание числа выполненных скобок для второго алгоритма.

Теорема 20

$$\max\{n_1, n_2\} \geq \frac{3}{4} \sum_j \hat{z}_j.$$

Доказательство Поскольку всегда $\max\{n_1, n_2\} \geq (n_1 + n_2)/2$, достаточно показать, что $(n_1 + n_2)/2 \geq \frac{3}{4} \sum_j \hat{z}_j$. Пусть S^k обозначает множество скобок, содержащих ровно k литералов. Имеем:

$$n_1 = \sum_k \sum_{C_j \in S^k} (1 - 2^{-k}) \geq \sum_k \sum_{C_j \in S^k} (1 - 2^{-k}) \hat{z}_j.$$

По лемме 3 имеем:

$$n_2 \geq \sum_k \sum_{C_j \in S^k} \beta_k \hat{z}_j.$$

Следовательно,

$$\frac{n_1 + n_2}{2} \geq \sum_k \sum_{C_j \in S^k} \frac{(1 - 2^{-k}) + \beta_k}{2} \hat{z}_j.$$

Простое вычисление показывает, что $(1 - 2^{-k}) + \beta_k \geq 3/2$ для всех натуральных k и, значит,

$$\frac{n_1 + n_2}{2} \geq \frac{3}{4} \sum_k \sum_{C_j \in S^k} \hat{z}_j = \frac{3}{4} \sum_j \hat{z}_j.$$

□

3.5 Дерандомизация и метод условных вероятностей.

Оказывается в некоторых случаях вероятностные алгоритмы могут быть "дерандомизированы", т.е. конвертированы в детерминированные алгоритмы. Метод, позволяющий сделать это, называется "методом условных вероятностей".

Проблемы, связанные с дерандомизацией вероятностных алгоритмов при построении хороших целочисленных решений в настоящее время находятся в центре внимания многих исследователей (см. [MR95]). После работы [Rag88], в которой вероятностным методом было доказано существование хороших целочисленных решений в задаче аппроксимации на решетке появилось много работ, в которых та же техника использовалась для других задач. Эта техника получила название метода *условных вероятностей*.

При этом подходе эффективный детерминированный алгоритм нахождения искомого целочисленного вектора можно построить путем аппроксимации исходной задачи некоторым функционалом (обычно связанным с оценками вероятностей некоторых событий) и затем использовать покоординатное построение требуемого

решения на основе вычислений проекций этого функционала. Опишем этот подход подробнее на примере задачи: имеется величина $X(x)$, где в булевом векторе $x = (x_1, \dots, x_n)$ компоненты являются независимыми случайными величинами, причем $P\{x_i = 1\} = p_i$, $P\{x_i = 0\} = 1 - p_i$. Так, в задаче MAX-SAT (задаче 15, рассмотренной в предыдущем разделе), $X(x_1, \dots, x_n)$ равно числу невыполненных скобок в КНФ при вероятностном округлении.

Требуется найти булев вектор \hat{x} , для которого выполнено неравенство

$$X(\hat{x}) \leq \mathbf{E}X. \quad (3.8)$$

Обозначим через $X(x| x_1 = d_1, \dots, x_k = d_k)$ новую случайную величину, которая получена из X фиксированием значений первых k булевых переменных.

Рассмотрим покомпонентную стратегию определения искомого вектора \hat{x} . Для определения его первой компоненты в $x = (x_1, \dots, x_n)$ вычисляем значения $f_0 \leftarrow \mathbf{E}X(x| x_1 = 0)$ и $f_1 \leftarrow \mathbf{E}X(x| x_1 = 1)$. Если $f_0 < f_1$, полагаем $x_1 = 0$, иначе полагаем $x_1 = 1$. При определенной таким образом первой компоненте (обозначим ее d_1) вычисляем значения функционала $f_0 \leftarrow \mathbf{E}X(x| x_1 = d_1, x_2 = 0)$ и $f_1 \leftarrow \mathbf{E}X(x| x_1 = d_1, x_2 = 1)$.

Если $f_0 < f_1$, полагаем $x_2 = 0$, иначе полагаем $x_2 = 1$.

Фиксируем вторую координату (обозначая ее d_2) и продолжаем описанный процесс до тех пор, пока не определится последняя компонента решения (см. Рис. 3.1).

Почему найденный вектор будет удовлетворять (3.8)? Рассмотрим первый шаг алгоритма. Имеем

$$\begin{aligned} \mathbf{E}X &= \mathbf{P}\{x_1 = 1\}\mathbf{E}X(x| x_1 = 1) + \mathbf{P}\{x_1 = 0\}\mathbf{E}X(x| x_1 = 0) \\ &= p_1\mathbf{E}X(x| x_1 = 1) + (1 - p_1)\mathbf{E}X(x| x_1 = 0) \\ &\geq p_1\mathbf{E}X(x| x_1 = d_1) + (1 - p_1)\mathbf{E}X(x| x_1 = d_1) \\ &= (p_1 + 1 - p_1)\mathbf{E}X(x| x_1 = d_1) = \mathbf{E}X(x| x_1 = d_1). \end{aligned}$$

Продолжая эту цепочку неравенств для каждого шага, получаем на последнем n -м шаге

$$\mathbf{E}X \geq \mathbf{E}X(x| x_1 = d_1, \dots, x_n = d_n).$$

Но

$$\mathbf{E}X(x| x_1 = d_1, \dots, x_n = d_n) = X(d_1, \dots, d_n),$$

и для построенного вектора $\hat{x} = (d_1, \dots, d_n)$ выполнено неравенство (3.8).

Как оценить число шагов описанной процедуры? На каждой итерации вычисляются два условных мат. ожидания (f_0 и f_1) и затем находится минимум этих величин. Если это делается за время $O(L)$, то при общем числе итераций n , получим

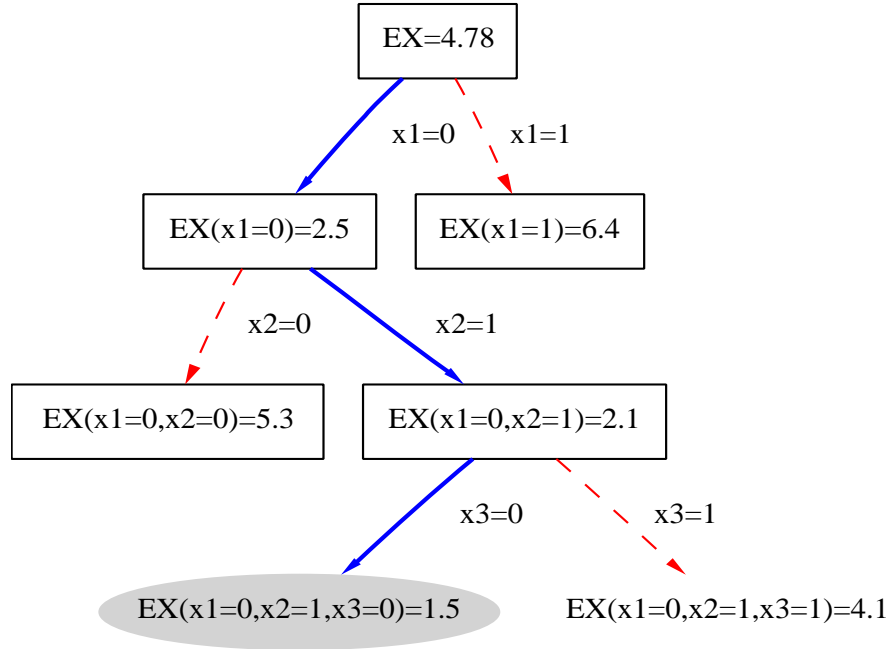


Рис. 3.1: Дерандомизация на основе минимизации оценок матожидания.

оценку $O(nL)$. Таким образом, изложенный общий метод позволяет осуществлять "дерандомизацию" если у нас есть эффективный алгоритм вычисления условных мат. ожиданий (или условных вероятностей).

Теперь используем описанный выше метод условных вероятностей для задачи 15 (MAX-SAT). Мат. ожидание X равно сумме вероятностей невыполнения скобок:

$$\mathbf{EX} = \sum_{j=1}^m P_j, \quad (3.9)$$

где вероятность невыполнения j -й дизъюнкции есть

$$P_j = \mathbf{P} \left\{ \sum_{i \in C_j^+} x_i + \sum_{i \in C_j^-} (1 - x_i) = 0 \right\}.$$

Для полученного вектора (d_1, \dots, d_n) выполнено неравенство

$$X(x_1 = d_1, \dots, x_n = d_n) \leq \mathbf{EX}. \quad (3.10)$$

Из теоремы 19 следует, что $\mathbf{EX} \leq (1/e)m$ и из (3.10) вытекает, что $X(x_1 = d_1, \dots, x_n = d_n) \leq (1/e)m$.

Таким образом, мы находим допустимый 0-1 вектор $\mathbf{x} = (x_1, \dots, x_n)$ с гарантированной верхней оценкой для целевой функции.

Важный вопрос заключается в том как эффективно вычислять условные мат. ожидания. Предположим, значения первых k переменных уже определены и I_0 — множество индексов переменных, значения которых равны 0, а I_1 — множество индексов переменных, значения которых равны 1.

Нетрудно проверить, что если $I_0 \cap C_j^- \neq \emptyset$ или $I_1 \cap C_j^+ \neq \emptyset$, то $P_j = 0$. В противном случае,

$$P_j = \prod_{i \in C_j^+ \setminus I_0} (1 - p_i) \cdot \prod_{i \in C_j^- \setminus I_1} p_i. \quad (3.11)$$

Запишем этот алгоритм более формально (алгоритм 20).

Алгоритм 20 Дерандомизация вероятностного алгоритма 19 по методу условных вероятностей

Вход: Формулировка задачи 15 в виде (3.6)

Выход: (y_1, \dots, y_m) — приближенное решение (3.6).

$(p_1, \dots, p_n) \leftarrow$ решения линейной релаксации (3.7)

for all $i \in \{1..m\}$ **do**

$f_0 = \mathbf{E}X(x \mid x_i = 0)$ {Вычисляется через (3.11)}

$f_1 = \mathbf{E}X(x \mid x_i = 1)$ { и (3.9)}

if $f_0 < f_1$ **then**

$x_i \leftarrow 0$

else

$x_i \leftarrow 1$

end if

end for

return \mathbf{x}

Упражнение 39 Покажите, что можно организовать вычисление f_0 и f_1 таким образом, что сложность алгоритма 20 будет $O(mn)$.

Глава 4

Приложения теории сложности.

4.1 Элементы криптографии с открытым ключом

Односторонние функции. Криптография с открытым ключом. Протокол выработки общего ключа. Свойства дискретного логарифма. Система RSA и ее анализ.

Центральным понятием "новой криптографии" является понятие односторонней функции.

Определение 40 Односторонней называется функция $F : X \rightarrow Y$, обладающая двумя свойствами:

1. существует полиномиальный алгоритм вычисления значений $F(x)$;
2. не существует полиномиального алгоритма инвертирования функции F (т.е. вычисления обратной функции — решения уравнения $F(x) = y$ относительно x).

Это не совсем формальное определение. Более формально свойство 2 формулируется так: любая полиномиальная вероятностная машина Тьюринга T может по данному y найти x из уравнения $F(x) = y$ лишь с пренебрежимо малой вероятностью (меньше $\frac{1}{p(n)}$ для любого полинома $p(n)$, где n — длина входа).

Вопрос о существовании односторонних функций пока открыт и является одним из центральных в теоретической криптографии и теории сложности. Однако, есть функции, которые предположительно являются односторонними. Одна из них — дискретный логарифм (уже упоминавшийся в разделе [1](#)).

4.1.1 Дискретный логарифм. Обмен ключами.

Задача 16 *Дискретный логарифм.*

Даны примитивный элемент g , b , простое число p . Найти x такое, что

$$g^x = b \pmod{p}.$$

Опишем сейчас применение дискретного логарифма для задачи формирования общего секретного ключа двумя пользователями (задача 17), связанными открытым (для противника) каналом связи.

Задача 17 *Абоненты A и B взаимодействуют по открытому каналу связи. Могут ли они, не имея вначале никакой секретной информации, организовать обмен так, чтобы в конце у них появлялся общий секретный ключ. Предполагается, что пассивный противник может перехватить все сообщения, которыми они обмениваются.*

Диффи и Хеллман предложили решать эту задачу с помощью дискретного логарифма (алгоритм 21).

Алгоритм 21 Протокол выработки общего секретного ключа

1. A и B независимо друг от друга выбирают по одному натуральному числу X_A и X_B . Эти элементы они держат в секрете.
2. Каждый из них вычисляет новый элемент

$$Y_A = a^{X_A} \pmod{p}, \quad Y_B = a^{X_B} \pmod{p},$$

причем числа p и a считаются общедоступными. Потом они обмениваются этими элементами по каналу связи.

3. A получив Y_B и зная свой секретный элемент X_A вычисляет новый элемент

$$Y_B^{X_A} = (a^{X_B})^{X_A} \pmod{p},$$

Аналогично поступает B :

$$Y_A^{X_B} = (a^{X_A})^{X_B} \pmod{p},$$

После этого у A и B появился общий элемент $a^{X_A X_B} \pmod{p}$, который и объявляется общим ключом.

Противник знает p , a , a^{X_A} , a^{X_B} , но не знает X_A и X_B , и хочет узнать $a^{X_A X_B}$.

В настоящее время нет алгоритмов решения этой задачи, более эффективных чем дискретное логарифмирование. А это — трудная математическая задача.

Дискретный логарифм обладает интересным свойством: если эта функция сложна в *худшем случае*, то сложна и в *среднем*. Доказательство этого свойства простое и мы приведем его сейчас, предварительно дав более точную формулировку.

Теорема 21 Пусть существует полиномиальный алгоритм A , который при случайном и равномерном выборе $b \in \mathbf{Z}_p$ правильно решает задачу дискретного логарифмирования на доле b не менее $1/n^{O(1)}$ (здесь n обозначает длину двоичной записи b).

Тогда существует полиномиальный по n вероятностный алгоритм, который находит дискретный логарифм для всех b .

Доказательство Построим новый вероятностный алгоритм B :

1. выберем x' случайно и равномерно из \mathbf{Z}_p
2. вычислим $b' = g^x g^{x'} \mod p$.

Заметим, что величина $g^{x'} \mod p$ равномерно распределена в \mathbf{Z}_p (докажите это в качестве упражнения!). Тогда b' — случайный элемент, равномерно распределенный в \mathbf{Z}_p (докажите и этот факт).

А далее все просто. Применим алгоритм A к b' : при этом мы получим ответ с вероятностью не менее $1/n^{O(1)}$.

А по нему очень легко восстановить ответ для исходной задачи, т.е. найти дискретный логарифм b . Действительно, дискретный логарифм от b' по определению равен $x + x'$, т.к. $g^x g^{x'} = g^{x+x'}$. Но мы знаем x' , поэтому легко можем найти x .

Далее используя стандартную технику амплификации (т.е. повторяя для данного b независимо эту процедуру полиномиальное число раз), можно сделать вероятность ошибки экспоненциально малой. \square

Упражнение 40 Выберем x случайно и равномерно из \mathbf{Z}_p . Докажите, что величина $g^x \mod p$ равномерно распределена в \mathbf{Z}_p .

Упражнение 41 Выберем x случайно и равномерно из \mathbf{Z}_p . Докажите, что для любого фиксированного b величина $b' = bg^x \mod p$ равномерно распределена в \mathbf{Z}_p .

4.1.2 Криптография с открытым ключом. Система RSA и ее анализ

Еще одним новым понятием криптографии является понятие *функции с секретом*.

Определение 41 Функцией с секретом K называется функция $F_K : X \rightarrow Y$, зависящая от параметра K и обладающая следующими свойствами:

1. при любом K существует полиномиальный алгоритм вычисления значений $F_K(x)$;
2. при неизвестном K не существует полиномиального алгоритма инвертирования F_K ;
3. при известном K существует полиномиальный алгоритм инвертирования F_K .

Существование таких функций также не доказано, но для практических целей криптографии было построено несколько функций, которые могут оказаться функциями с секретом. Для них свойство 2 пока строго не доказано, но считается, что задача инвертирования эквивалентна некоторой давно изучаемой трудной (с алгоритмической точки зрения) математической задаче. Наиболее известной и популярной из них является теоретико-числовая функция, на которой построен шифр RSA.

Сначала приведем общую схему *криптосистемы с открытым ключом*.

1. Пользователь A , который хочет получать зашифрованные сообщения, должен выбрать какую-нибудь функцию F_K с секретом K .
2. Он публикует описание функции F_K в качестве своего алгоритма шифрования. При этом значение секрета K он никому не сообщает и держит в секрете.
3. Пользователь B посылает пользователю A защищаемую информацию $x \in X$, вычисляя $y = F_K(x)$ и посылая y по открытому каналу пользователю A .
4. Поскольку A знает секрет K , то он умеет эффективно инвертировать F_K . Он вычисляет x по полученному y . Никто другой не знает K и, поэтому, в силу свойства 2 функции с секретом, не сможет за полиномиальное время по зашифрованному сообщению $F_K(x)$ вычислить защищаемую информацию x .

Схема RSA устроена следующим образом (алгоритм [22](#)).

Докажем сейчас однозначность декодирования.

Теорема 22 Для всех x

$$x^{ed} = x \pmod{n}.$$

Доказательство Имеем:

$$ed = 1 \pmod{\phi(n)},$$

откуда получаем, что для некоторого k

$$ed = 1 + k(p-1)(q-1).$$

Алгоритм 22 Схема RSA

1. Выбирают два достаточно больших простых числа p и q (обычно около 100 десятичных знаков).
2. Находят $n = pq$.
3. Выбирают число e , взаимно-простое с $p - 1$ и $q - 1$:

$$\text{нод}(e, p - 1) = \text{нод}(e, q - 1) = 1.$$

Здесь и далее $\text{нод}(a, b)$ обозначает наибольший общий делитель чисел a и b .

4. Вычисляют функцию Эйлера $\phi(n)$, равную числу натуральных чисел не превосходящих n и взаимно-простых с ним по формуле:

$$\phi(n) = \phi(p)\phi(q) = (p - 1)(q - 1).$$

5. Находят целое число d такое, что

$$de \equiv 1 \pmod{\phi(n)}, \quad 1 \leq d < \phi(n).$$

Такое число существует и единственно, поскольку $\text{нод}(e, \phi(n)) = 1$.

- Функция F_K , реализующая шифрование в схеме RSA устроена следующим образом:

$$F_K : x \rightarrow x^e \pmod{n}.$$

- Дешифрование осуществляется функцией

$$G(a) = a^d \pmod{n}.$$

Секретом является число p (или q) в разложении n на простые множители (или d).

Пусть $x \neq 0$. Малая теорема Ферма гласит, что $x^{p-1} = 1 \pmod p$, откуда мы получаем, что

$$\begin{aligned} x^{ed} &= x(x^{p-1})^{k(q-1)} \pmod p \\ &= x(1)^{k(q-1)} \pmod p \\ &= x \pmod p. \end{aligned}$$

Если же $x = 0$, то тривиально имеем $x^{ed} = x \pmod p$.

Аналогично рассматриваем соотношения по модулю q и получаем, что для всех x

$$x^{ed} = x \pmod q.$$

Теперь остается применить китайскую теорему об остатках, которая гласит: для взаимно-простых чисел r_1, \dots, r_k любое число $0 \leq n < r_1 \cdot \dots \cdot r_k$ однозначно восстанавливается по остаткам

$$n = n_1 \pmod{r_1}, n = n_2 \pmod{r_2}, \dots, n = n_k \pmod{r_k}.$$

По китайской теореме об остатках имеет место соотношение:

$$x^{ed} = x \pmod{pq}.$$

□

Упражнение 42 Система RSA является мультипликативной в том смысле, что произведение кодов равно коду произведения

$$x^e y^e \pmod n = (xy)^e \pmod n.$$

Используя этот факт докажите, что если противник имеет алгоритм расшифровки 1 процента сообщений случайно выбранных из \mathbf{Z}_n , то он может переработать его в вероятностный алгоритм расшифровки произвольного сообщения, закодированного кодом RSA, с большой вероятностью (аналогично теореме [21](#)).

4.2 Коммуникационная сложность.

Коммуникационная сложность. Задача сравнения идентичности битовых строк

С появлением и развитием телекоммуникационных сетей и распределенных вычислений, стали возникать новые типы ресурсных ограничений: коммуникационные. О них не могло быть и речи, пока компьютеры были вне сети — хватало рассмотренных в разделе [1](#) временных и пространственных ресурсных ограничений и соответствующих мер сложности задач. Объединение компьютеров в локальные сети также редко приводило к тому, что "бутылочным горлышком" при решении какой-либо задачи является именно пропускная способность сети, или высокая стоимость

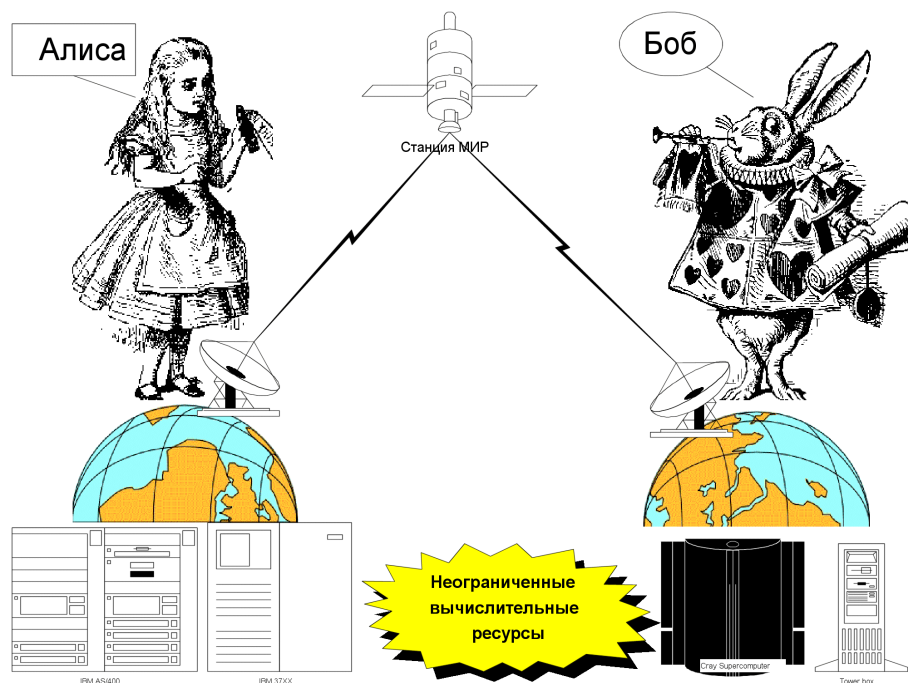


Рис. 4.1: Коммуникационная сложность задачи для двух участников.

траффика. Однако, с появлением глобальных сетей стали создаваться ситуации, когда, например, несколько мощных научных центров, обладающих огромными вычислительными ресурсами, пытаются объединить усилия для решения некоторой вычислительной задачи, либо когда филиалам транснациональной корпорации требуется выполнить возможно алгоритмически несложные действия, но над распределенными базами данных, при этом основной стоимостью становится стоимость коммуникаций. Как напрямую, путем оплаты услуг провайдера, так опосредовано — когда передача больших объемов данных занимает много дорогостоящего времени и тормозит вычисления.

Таким образом, появилось новое ресурсное ограничение — *стоимость коммуникации*, и соответственно появилась новая мера сложности алгоритмических задач — *коммуникационная сложность*.

При исследовании коммуникационной сложности задачи полагают, что входные данные некоторым образом распределены между $n > 1$ участниками, каждый из которых обладает неограниченными вычислительными ресурсами, и необходимо установить нижние и верхние оценки для траффика (объема сообщений), необходимого для решения задачи. Существуют различные модели коммуникации, обуславливающие различные меры коммуникационной сложности: модели с произвольным числом участников, модели с произвольным распределением данных и т.п. Наиболее стандартной и классической является модель с двумя участниками и симметричным распределением между ними входных данных¹ (См. Рис. 4.1). Их задача —

¹Как и в многих задачах связанных с коммуникациями (например, в криптографических постановках) этих двух участников зовут Алиса и Боб.

совместно вычислить некоторую булеву функцию. Участники по очереди обмениваются сообщениями фиксированной длины (вариант — однобитовыми сообщениями), пока одним из участников не будет получен ответ. Коммуникационная сложность оценивается в терминах $O(f(n))$, где n — длина входа, (например, число аргументов вычисляемой булевой функции). Сразу заметим, что для любой задачи такого рода существует тривиальное решение, заключающееся в пересылке одним участком другому всех своих данных — $O(n)$. Это является верхней оценкой коммуникационной сложности для задач с двумя участниками, поэтому интерес составляет получение более низких оценок — полилогарифмических или константных.

Приведем примеры нескольких задач, с установленными для них оценками коммуникационной сложности.

Задача 18 ЧЕТНОСТЬ

Алиса и Боб имеют по битовой строке длины n . Необходимо вычислить четность числа бит строки-конкатенации.

Легко видеть, что коммуникационная сложность задачи 18 равна $O(1)$. Действительно, Алисе достаточно вычислить четность своей строки (один бит), и передать ее Бобу.

Задача 19 СРАВНЕНИЕ

Алиса и Боб имеют битовые строки длины n : $X = (x_1, \dots, x_n)$ и $Y = (y_1, \dots, y_n)$ соответственно.

Нужно сравнить эти строки (проверить на эквивалентность).

Оказывается (доказано), что нижние оценки для задачи 19 совпадают с тривиальными верхними оценками, и таким образом, коммуникационная сложность точного решения задачи 19 есть $\Omega(n)$.

Задача 20 СУММА БИТ

Алиса и Боб имеют по битовой строке длины n . Одинаково ли число единиц в битовых строках?

Простой алгоритм для задачи 20 — Алиса высылает Бобу сумму единиц в своей строке, имеет коммуникационную сложность $O(\log n)$. С другой стороны, доказано (здесь мы не будем касаться методик получения нижних оценок), что этот алгоритм также является оптимальным.

Вот задача (правда вычисляется не булевая функция, а число), требующая менее тривиального оптимального алгоритма:

Задача 21 МЕДИАНА

Алиса и Боб имеют по подмножеству множества $(1, 2, \dots, n)$. Необходимо найти элемент-медиану² в объединении этих подмножеств.

Упражнение 43 Постройте алгоритм для задачи 21 с коммуникационной сложностью $O(\log^2 n)$.

Упражнение 44 Постройте алгоритм для задачи 21 с коммуникационной сложностью $O(\log n)$.

Кроме коммуникационной сложности точного решения задачи и детерминированных алгоритмов рассматривают (аналогично с разделом 3) коммуникационную сложность вероятностных алгоритмов. Также, как и в случае с временной сложностью, применения вероятностных алгоритмов может дать существенный выигрыш и в коммуникационной сложности.

Итак, рассмотрим задачу 19 (СПРАВНЕНИЕ), и попробуем получить алгоритм, более коммуникационно-эффективный, чем тривиальный.

Выберем случайное простое p из интервала $[1 \dots M]$, M - целое. Обозначим

$$p(X) = X \bmod p, \quad p(Y) = Y \bmod p.$$

Очевидно, что если строки X и Y совпадают, то $p(X) = p(Y)$. Ошибка произойдет, если $|X - Y|$ делится на p без остатка. Оценим вероятность ошибки.

Обозначим через $\pi(N)$ число простых, не превосходящих N . Известна следующая оценка: $\pi(N) \sim \frac{N}{\ln N}$. Так как длина битовых строк равна n , то $|X - Y| \leq 2^n$.

Лемма. Число различных простых делителей любого числа меньшего 2^n не превосходит n .

Пусть $A(n)$ -число простых, делящих $X - Y$. Тогда вероятность ошибки равна отношению числа простых, делящих $X - Y$, к числу всех простых из интервала $[1 \dots M]$, то есть

$$P_{err} = \frac{A(n)}{\pi(M)} \leq \frac{\pi(n)}{\pi(M)} = \frac{n \ln M}{M \ln n},$$

причем $M > n$.

Для того, чтобы вероятность ошибки была мала, M должно быть достаточно велико. Пусть $M = n^c$, $c = \text{const}$, тогда

$$P_{err} = \frac{nc \ln n}{n^c \ln n} = \frac{c}{n^{c-1}}.$$

Видно, что при больших n , P_{err} очень мала.

²элемент, занимающий $\lceil m/2 \rceil$ место в упорядоченном массиве из m элементов

Таким образом, для сравнения двух строк достаточно переслать не более $\log M = c \log n$ бит.

Пример:

$$n = 100000 \text{ бит} \rightarrow M = 2^{32} = 4 * 10^9 \rightarrow P_{err} = 10^{-4}.$$

Глава 5

Глоссарий и основные определения

По результатам чтения лекций, были выявлены основные матобозначения вызывающие у студентов затруднения. Они приведены в конце курса, как справочный материал.

Итак,

Определение 42 Алфавит — конечный набор символов.

Определение 43 Слово — конечная последовательность символов из некоторого алфавита Σ .

Пустое слово обозначается \emptyset . Набор слов длины n над алфавитом Σ обозначается Σ^n , набор всех слов (включая пустое) — Σ^* .

Определение 44 Язык L — произвольное подмножество $L \subseteq \Sigma^*$, т.е. произвольное множество слов над алфавитом Σ .

Надо различать пустой язык (язык не содержащий ни одного слова), который также обозначается \emptyset , от языка $\{\emptyset\}$, содержащего единственное пустое слово.

Определение 45 Дополнение языка L — язык \bar{L} , состоящий из всех возможных слов над алфавитом Σ , не входящих в язык L — $\bar{L} \equiv \Sigma^* \setminus L$.

Множество вещественных чисел обозначается R , целых — Z , рациональных — Q , натуральных — N . Неотрицательные их подмножества обозначаются соответственно R_+ , Z_+ , Q_+ .

Пусть $f : N \rightarrow R$, $g : N \rightarrow R$.

Определение 46 $f = O(g)$, если $\exists c \in R, c > 0$, и $\exists n_0 \in N, \forall n > n_0 : |f(n)| \leq c|g(n)|$.

Определение 47 $f = o(g)$, если f равно нулю лишь в конечном числе точек и $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Определение 48 $f = \Omega(g)$, если $g = O(f)$.

Определение 49 $f = \Theta(g)$, если $f = O(g)$ и $g = O(f)$.

Определение 50 **Кортеж** (k -кортеж) — упорядоченный набор из k элементов (множеств, подмножеств, элементов множеств). Обозначается угловыми скобками, например кортеж из элементов a , b , и множества C — $\langle a, b, C \rangle$.

5.1 Список Упражнений

Упражнение 1 Докажите, что существуют невычислимые по Тьюрингу, функции $y = f(x)$. Использовать мощностные соображения.

Указание: рассмотреть мощность множества машин Тьюринга и мощность множества булевых функций $f : N \rightarrow \{0, 1\}$

Упражнение 2 Докажите, что также неразрешима версия задачи 5, остановка на пустом слове. А именно, для данной МТ T , определить, остановится ли она на пустом слове.

Упражнение 3 Докажите, что неразрешима Проблема недостижимого кода — существует ли алгоритм, который для заданной машины Тьюринга T , и ее состояния q_k отвечает на вопрос, попадет ли машина в это состояние хотя бы для одного входного слова x (или это состояние недостижимо).

Упражнение 4 Докажите, что не существует алгоритма, который выписывает одну за другой все машины Тьюринга, которые не останавливаются, будучи запущенными на пустой ленте.

Упражнение 5 Существует ли алгоритм, который выписывает одну за другой все машины Тьюринга, которые останавливаются, будучи запущенными на пустой ленте?

Упражнение 6 Покажите $PSPACE \subseteq EXPTIME$.

Упражнение 7 Пусть c_n есть максимум сложности $s(f)$ по всем булевым функциям f от n переменных. Докажите, что $1,99^n < c_n < 2,01^n$ при достаточно больших n .

Упражнение 8 Покажите, что любую функцию можно вычислить схемой глубины не более 3 из элементов *NOT* и из элементов *AND* и *OR* с произвольным числом входов.

Упражнение 9 Докажите, что существует разрешимый предикат, который принадлежит $P/poly$, но не принадлежит P .

Упражнение 10 Покажите, что $P \subseteq NP \cap co-NP$.

Упражнение 11 Преобразуйте отношение ЭКВИВАЛЕНТНОСТЬ в 3-КНФ.

Упражнение 12 Покажите, что задача распознавания гамильтоновых графов (т.е. графов, содержащих гамильтонов цикл), принадлежит **NP**.

Упражнение 13 Покажите, что задача распознавания негамильтоновых графов (т.е. графов, не содержащих ни одного гамильтонова цикла), принадлежит **Co-NP**.

Упражнение 14 Постройте пример, где оценка $O(1 + \ln t)$ размера покрытия построенного жадным алгоритмом достигается по порядку. Указание: достаточно рассмотреть случай, когда размер минимального покрытия $M = 2$.

Упражнение 15 Покажите, что сложность алгоритма 11 (нахождение эйлерова обхода) есть $O(|E|)$.

Упражнение 16 Рассмотрим модификацию задачи СУММА РАЗМЕРОВ, разрешим даже отрицательные размеры. Формально: Даны натуральные числа a_i , $\forall i \in [1 \dots n]$ — $n^2 \leq a_i \leq n^2$, и число A . Существует ли решение в 0-1 переменных (x_1, \dots, x_n) уравнения $\sum_{i=1}^n a_i x_i = A$?

Существует ли полиномиальный алгоритм для этой задачи?

Упражнение 17 Для значения решения f_G полученного модифицированным жадным алгоритмом для задачи о рюкзаке, и оптимального значения f^* выполняется

$$f^*/2 \leq f_G \leq f^*. \quad (5.1)$$

Упражнение 18 Имеется квадратная, $n \times n$ матрица A , элементами которой являются линейные функции $f_{ij}(x) = a_{ij}x + b_{ij}$.

Придумайте Монте-Карло алгоритм с односторонней ошибкой, для проверки этой матрицы на вырожденность ($\det A \equiv 0$).

Упражнение 19 Докажите эквивалентность задачи MAX-SAT (задача 15) и ЦЛП (3.6).

Упражнение 20 Выберем x случайно и равномерно из \mathbf{Z}_p . Докажите, что величина $g^x \bmod p$ равномерно распределена в \mathbf{Z}_p .

Упражнение 21 Выберем x случайно и равномерно из \mathbf{Z}_p . Докажите, что для любого фиксированного b величина $b' = bg^x \bmod p$ равномерно распределена в \mathbf{Z}_p .

Упражнение 22 Постройте алгоритм для задачи [21](#) с коммуникационной сложностью $O(\log n)$.

Список таблиц

1.1	Значения функции $n!$	9
-----	---------------------------------	---

Список иллюстраций

1.1	Иллюстрация типичных исходных данных для задачи КОММИВОЯ-ЖЕР	8
1.2	Иллюстрация работы алгоритма 6	14
1.3	Трехленточная МТ в действии	26
1.4	Трехленточная универсальная МТ для одноленточных МТ	29
1.5	Схема сравнения двух битовых строк	41
1.6	Иерархия классов P, NP, coNP.	56
3.1	Дерандомизация на основе минимизации оценок матожидания.	90
4.1	Коммуникационная сложность задачи для двух участников.	98

Список алгоритмов

1	Тривиальное вычисление $y = x^n \bmod m$	3
2	Разумное вычисление $y = x^n$	4
3	Вычисление факториала $y = n! \bmod m$	5
4	Переборный алгоритм для КОММИВОЯЖЕР	9
5	Алгоритм КРАТЧАЙШИЙ ПУТЬ В ГРАФЕ в одну вершину	11
6	Упрощенный алгоритм Прима	13
7	Быстрое переполнение памяти в присутствии умножения.	24
8	Простой способ вычисления произведения $R_1 \cdot R_2$ на RAM.	24
9	Сведение оптимизационных задач к задачам разрешения	47
10	Моделирование перебора недетерминизмом	50
11	Алгоритм нахождения эйлерава обхода	60
12	Простой алгоритм для решения метрической задачи коммивояжера . .	62
13	Алгоритм Кристофидеса для решения метрической TSP	64
14	Динамическое программирование для задачи СУММА РАЗМЕРОВ . .	65
15	Динамическое программирование для задачи 0-1 РЮКЗАК	68
16	ε -оптимальный алгоритм для задачи 0-1 РЮКЗАК	70
17	Улучшенный ε -оптимальный алгоритм для задачи 0-1 РЮКЗАК . . .	72
18	Полиномиальный в среднем алгоритм для задачи об упаковке	75
19	Приближенный вероятностный алгоритм для задачи 15 (MAX-SAT) на основе линейной релаксации	86
20	Дерандомизация вероятностного алгоритма 19 по методу условных вероятностей	91
21	Протокол выработки общего секретного ключа	93
22	Схема RSA	96

Литература

- [М. 82] Д. С. Джонсон М. Гэри. *Вычислительные машины и труднорешаемые задачи*. М.: Мир, 1982. [52](#), [54](#)
- [А. 99] М. Вялый А. Китаев, А. Шень. *Классические и квантовые вычисления*. издательство МЦНМО, 1999. [40](#)
- [Дж.67] Р. Стернз Дж. Хартманис. О вычислительной сложности алгоритмов. In *Кибернетика, нов. сер.*, number 4, pages 57–85. М., Мир, 1967. [46](#)
- [Ред71] Н. П. Редькин. О минимальной реализации линейной функции схемой из функциональных элементов. *Кибернетика*, 6:31–38, 1971. [56](#)
- [Кар75a] Р. М. Карп. Сводимость комбинаторных задач. In ДАН СССР, editor, *Кибернетика, нов. сер.*, number 12, pages 16–38, 1975. [51](#), [52](#), [54](#)
- [Кук75b] С. А. Кук. Сложность процедур вывода теорем. In М. Мир, editor, *Кибернетика, нов. сер.*, number 12, pages 5–15, 1975. [47](#), [52](#), [53](#)
- [Е.М84] Е. Мендельсон. *Введение в математическую логику*. М. Наука, гл. ред. физ. мат. лит., 1984. [31](#)
- [Раз85a] А. А. Разборов. Нижние оценки монотонной сложности логического перманента. *Математические Заметки*, 37(4):887–900, 1985. [57](#)
- [Раз85b] А. А. Разборов. Нижние оценки монотонной сложности некоторых булевых функций. *ДАН СССР*, 281(4):798–801, 1985. [57](#)
- [Раз85c] А. А. Разборов. Нижние оценки размера схем ограниченной глубины в полном базисе, содержащем функцию логического сложения. *Математические Заметки*, 37(6), 1985. [57](#)
- [А.91] Схрейвер А. *Теория линейного и целочисленного программирования*. М. Мир, 1991. [37](#)
- [Н.Н96] А.А. Разборов Н.Н. Кузюрин. Оценка состояния и прогнозные исследования эффективных алгоритмов для точного и приближенного решения переборных задач дискретной оптимизации. Отчет по НИР., 1996. [2](#)
- [Том99] Рональд Ривест Томас Кормен, Чарльз Лейзерсон. *Алгоритмы: построение и анализ*. М.: МЦНМО, 1999. [13](#)

- [Adl78] Leonard M. Adleman. Two theorems on random polynomial time. In *IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978. [84](#)
- [AS92] N. Alon and J.H. Spencer. *The Probabilistic Method*. Wiley, 1992. [78](#)
- [Blu67] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967. [38](#)
- [Joh90] David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161. 1990. [38](#), [74](#)
- [L.G79] L.G.Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979. [37](#)
- [Lov] László Lovász. Complexity of algorithms. [2](#)
- [MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge Univ. Press, 1995. [81](#), [82](#), [84](#), [88](#)
- [Rag88] Prabhakar Raghavan. Probabilistic constructions of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Science*, 37(4):130–143, 1988. [88](#)
- [RW90] Ran Raz and Avi Wigderson. Monotone circuits for matching require linear depth. In *ACM Symposium on Theory of Computing*, pages 287–292, 1990. [57](#)
- [Sma00] Smale. Mathematical problems for the next century. In V. Arnold, M. Atiyah, P. Lax, and B. Mazur, editors, *Mathematics: Frontiers and Perspectives, IMU, AMS, 2000*. 2000. [4](#), [52](#)
- [SS71] A. Schonhage and V. Strassen. Schnelle multiplikation grosser zahlen, 1971. [35](#)
- [SV77] R. Solovay and Strassen V. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977. [84](#)